A MARSHALL CAVENDISH 10 COMPUTER COURSE IN WEEKLY PARTS

INPUT

LEARN PROGRAMMING - FOR FUN AND THE FUTURE

UK £1.00 Republic of Ireland £1.25 Malta 85c Australia $2.25 New Zealand $2.95

INPUT

Vol. 1 No 10

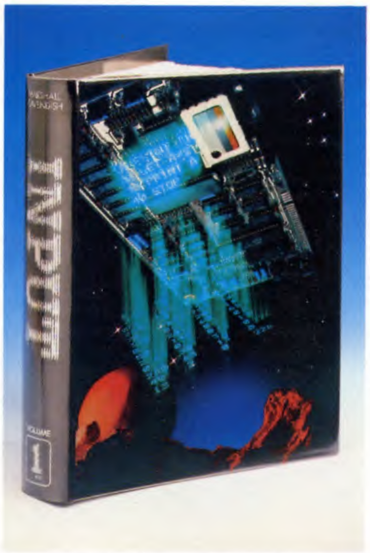
INDEX

The last part of INPUT, Part 52, will contain a complete, cross-referenced index. For easy access to your growing collection, a cumulative index to the contents of each issue is contained on the inside back cover.

PICTURE CREDITS

Front Cover, Science Photo Library/NASA Paul Chave. Page 289, Howard Kingsnorth. Pages 290, 291, 292, Nick Mijnheer. Page 294, Graeme Harris. Page 295, Nick Mijnheer. Pages 296, 298, 301, Alan Baker. Page 300, Bernard Fallon. Pages 302, 304, 306, Digital Arts. Pages 305, 306, Ray Duns. Pages 309, 310, 312, Chris Lyons. Pages 314, 317, Paul Chave. Pages 318–320, Science Photo Library/Paul Chave.

Published by Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA, England. Typeset by MS Filmsetting Limited, Frome, Somerset. Printed by Artisan Press, Leicester and Howard Hunt Litho, London.



There are four binders each holding 13 issues.

HOW TO ORDER YOUR BINDERS

UK and Republic of Ireland: Send £4.95 (inc p & p) (IR£5.45) for each binder to the address below:

Marshall Cavendish Services Ltd,
Department 980, Newtown Road,
Hove, Sussex BN3 7DN

Australia: See inserts for details, or write to INPUT, Gordon and Gotch Ltd, PO Box 213, Alexandria, NSW 2015

New Zealand: See inserts for details, or write to INPUT, Gordon and Gotch (NZ) Ltd, PO Box 1595, Wellington

Malta: Binders are available from local newsagents.

BACK NUMBERS

Copies of any part of INPUT can be obtained from the following addresses at the regular cover price, with no extra charge for postage and packing:

UK and Republic of Ireland:
INPUT, Dept AN, Marshall Cavendish Services,
Newtown Road, Hove BN3 7DN

Australia, New Zealand and Malta:
Back numbers are available through your local newsagent

COPIES BY POST

Our Subscription Department can supply your copies direct to you regularly at £1.00 each. For example the cost of 26 issues is £26.00; for any other quantity simply multiply the number of issues required by £1.00. These rates apply anywhere in the world. Send your order, with payment to:

Subscription Department, Marshall Cavendish Services Ltd,
Newtown Road, Hove, Sussex BN3 7DN

Please state the title of the publication and the part from which you wish to start.

HOW TO PAY: Readers in UK and Republic of Ireland: All cheques or postal orders for binders, back numbers and copies by post should be made payable to:
Marshall Cavendish Partworks Ltd.

QUERIES: When writing in, please give the make and model of your computer, as well as the Part No., page and line where the program is rejected or where it does not work. We can only answer specific queries – and please do not telephone. Send your queries to INPUT Queries, Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA.

INPUT IS SPECIALLY DESIGNED FOR:

The SINCLAIR ZX SPECTRUM (16K, 48K, 128 and +), COMMODORE 64 and 128, ACORN ELECTRON, BBC B and B+, and the DRAGON 32 and 64.

In addition, many of the programs and explanations are also suitable for the SINCLAIR ZX81, COMMODORE VIC 20, and TANDY COLOUR COMPUTER in 32K with extended BASIC. Programs and text which are specifically for particular machines are indicated by the following symbols:

SPECTRUM 16K, 48K, 128, and +

COMMODORE 64 and 128

ACORN ELECTRON, BBC B and B+

DRAGON 32 and 64

ZX81

VIC 20

TANDY TRS80 COLOUR COMPUTER

A COMPUTER TYPING TUTOR

- WHY LEARN TO TYPE?
- LEARNING THE HOME KEYS
- HOW TO USE THE PROGRAM
- IMPROVING YOUR SPEED AND ACCURACY

Whether you're going to sit down and write your first novel, or just want to speed up your copying of programs, fast, accurate typing is an essential first step

The fifth generation of computers promises far more direct communication between machine and user—with things like touch and voice control becoming commonplace. Already, some of these newer control systems are finding their way onto relatively low-priced business and industrial machines.

Unfortunately for the home computer enthusiast, however, it looks like being some time before such facilities are available on domestic machines, and for the moment, we are stuck with the keyboard.

Actually, the keyboard is a reasonably efficient method of giving the computer information or instructions—but its main disadvantage is that it is very time-consuming, particularly if you are not an expert (and *very* accurate) typist. And nothing is more frustrating than trying to copy a printed listing if you continually have to look from keyboard to screen to paper to ensure that your typing into the computer is accurate.

So why not use your computer to teach you to improve your typing? It's an ability that is useful not just for writing programs, but will also prove invaluable if you ever want to make use of the computer's potential as a word processor. The program which follows will help you get more familiar with the keyboard, and—if you don't cheat—increase your speed and your accuracy, too.

Spectrum owners are at a disadvantage here, because the Spectrum keyboard is by its nature less easy to type on at speed. But of course, for typing programs, you only need a single keystroke for each BASIC command.

ABOUT THE PROGRAM

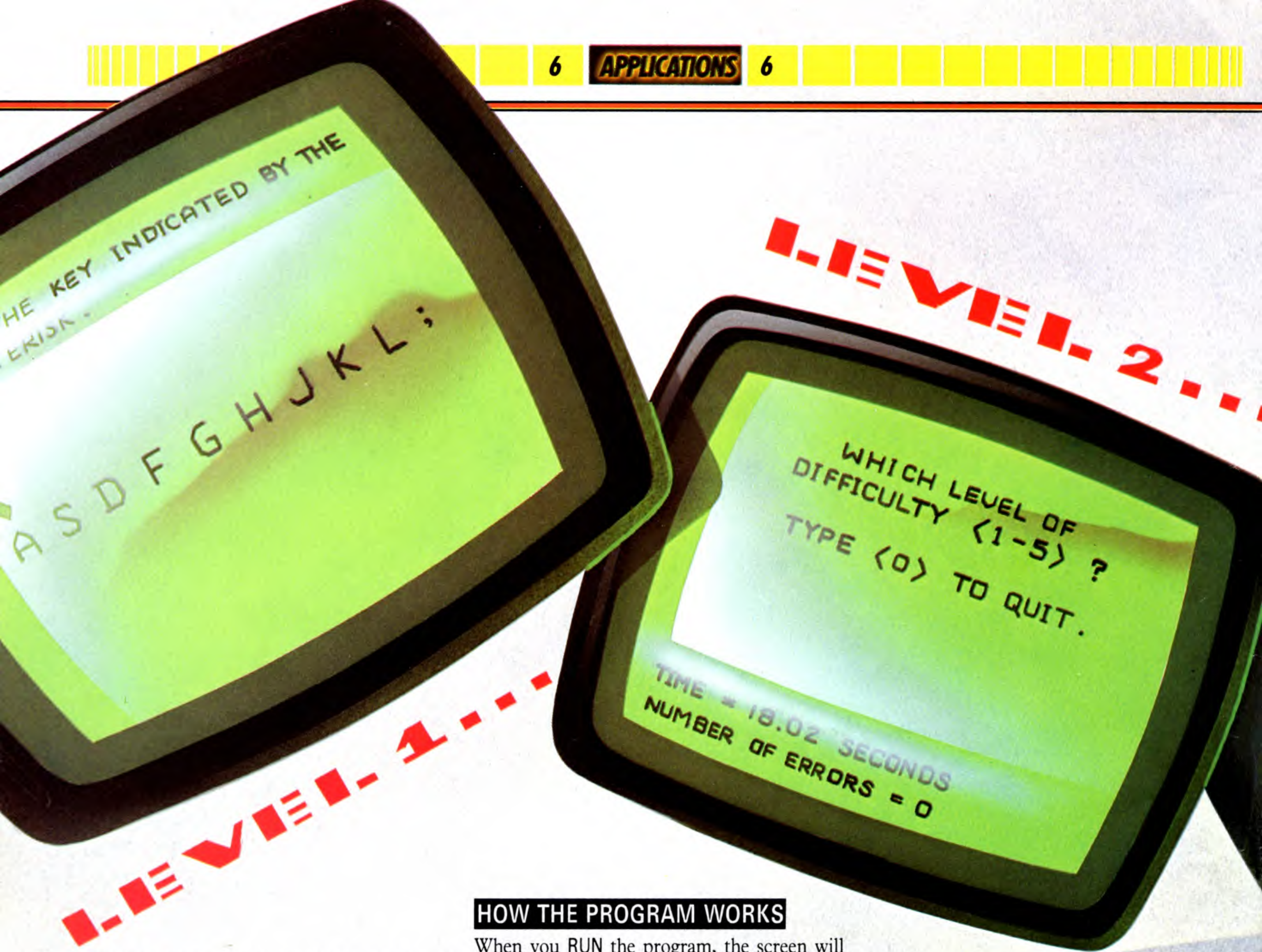
Just like a proper typing course which could cost a small fortune, the *INPUT* typing course is designed to build your skills up methodically, so that you will get results at every stage as you progress. But it's not a magic process, and like anything else worth learning, it takes time.

So the course is broken down into a series of easy stages, with add-on programming sections for each stage. In this way, when you are ready to move on, you can build up the existing program to take you on to the next level of proficiency. And as you go, you'll get a constant update on your speed and accuracy.

LEARNING THE 'HOME' KEYS

Real typing proficiency means being able to hit the right keys every time without having constantly to look at the keyboard. It's easier than it sounds, but you can only achieve it by starting the right way, and by keeping up the good habits once you have learned them.

Obviously, if you are going to be able to hit the right keys only by feel, you'll have to learn where they are. And just as importantly, you will need to ensure that your hands are always in the same position over the keyboard—otherwise you have no way of knowing where you are about to press down on it. Professional typists do this by ensuring that their fingers are always positioned over the same keys—called the 'home' keys. These are the keys in the middle row, that begins ASDF.... So these are the keys that you will learn first. Once you



```
HE KEY INDICATED BY THE
ERISK.

ASDFGHJKL;
```

```
WHICH LEVEL OF
DIFFICULTY (1-5) ?
TYPE (0) TO QUIT.

TIME = 18.02 SECONDS
NUMBER OF ERRORS = 0
```

master the home keys, the rest of the keyboard can be reached by moving your fingers only a short distance each time.

To get the maximum benefit from the course, you should try to stick to this and not cheat. But even if you are a 'peek and pick' one-finger typist, you should be able to see your skills improve.

When you have typed in the program—a task that should get easier as you go on with the course—and RUN it, sit with your hands poised over the keyboard ready to start, following the screen prompts which are explained below. The little finger of your left hand should be over the A key, the next finger over the S, and so on, with your index finger over the F. Your right index finger should be over the J, the middle finger over the K, and so on. Your thumbs (except on the Spectrum) should be poised over the space bar. This of course leaves two keys—G and H—uncovered in the middle. G can be reached with your left index finger and H with your right index finger.

If this sounds complicated, don't worry. The first part of the program is designed to make it familiar.

HOW THE PROGRAM WORKS

When you RUN the program, the screen will ask you to choose between five options. You should start with lesson 1 and master this before progressing to lesson 2 and so on.

Level 1

The screen displays the home keys in a line as they appear on the keyboard. An asterisk will appear above each key in turn, working from A to ; (A to : on the Commodore keyboard and A to L on the Spectrum). Strike each key in sequence as the asterisk moves over it, using the correct finger—remember to type G and H by moving your index fingers *only*, and use your right little finger for ; or :.

The asterisk will wait for you if you make a mistake. A correctly-hit key is marked by a beep, with a buzz for a missed or incorrect letter.

At the end of the exercise, the screen will display your time, and the number of errors you made.

Level 2

When you are familiar with the position of the keys, move on to this level which is similar to the first, except that now the asterisk moves from letter to letter at random—don't cheat by looking down! Once again, your time taken to complete the exercise (you get 20 random letters each go), and the number of mistakes you made, is displayed when you finish.

Level 3

This is similar to Level 2 in that you are given random letters, one at a time. But this time, the screen does not show you their position on the keyboard. Instead, they are just printed up singly on the centre of the screen. Once again, you get 20 each go.

Level 4

At last, you can type some words! All the words in this exercise are made up of letters on the home keys only—GLAD, for example. As in the previous level, 20 words appear at random in the centre of the screen, one after another. As you copy each word, an asterisk indicates each letter. And once more you get a score for speed and accuracy. Don't cheat by looking, or you will be handicapped for some of the more challenging tests which are to come later on in the course!

Level 5

This time a 'sentence' of 6 or 7 random words is printed out. As you type in the letters, they appear one by one under the printed words. When you have typed the whole line, you'll be told your score and an equivalent speed in words per minute (wpm). This is based on an average length of 5 letter words and a space between each word. When you are typing in the words that appear on the screen, don't forget the spaces, which you obtain with your thumbs on the space bar—on the Spectrum you need to press the awkwardly-sited SPACE key with your little finger.

Practise each of the levels until you can do them easily. Next, we add the remaining keys and some more complicated words.

LEVEL 3.
TYPE THE LETTER SHOWN ON THE SCREEN.
H
LEVEL 4...
TYPE THE WORD THAT APPEARS
DAD
TY

Q+A

What typing speed should I aim at to start with?

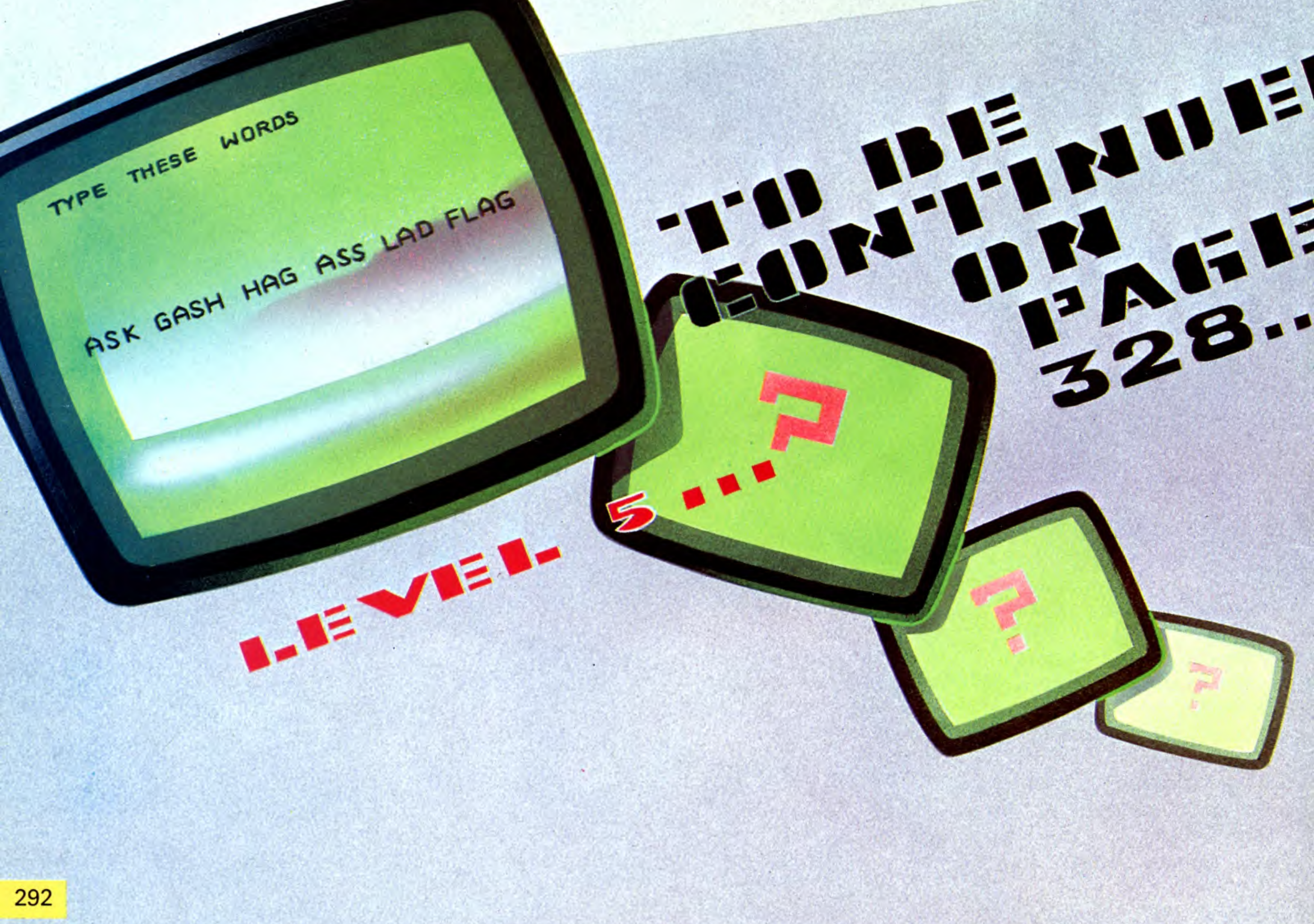
At this early stage of the course you should aim for accuracy rather than speed. Once you can type in all the letters and words without any errors you can try speeding up.

You should practice Levels 1, 2 and 3 until you can type the 20 letters in about 12 or 13 seconds, and for Levels 4 and 5 you should aim for about 15 words per minute or more.

```
10 BORDER 1: PAPER 1: INK 7: CLS
20 POKE 23658,8: LET ER=0
30 LET S$=“ASDFGHJKL”
100 PRINT INVERSE 1;AT 8,9; “□TYPING
   TUTOR□”
110 PRINT ''TAB 6;“SELECT LEVEL (1 TO 5)”
120 IF INKEY$=“” THEN GOTO 120
130 LET A$=INKEY$: IF A$<“1” OR
   A$>“5” THEN GOTO 120
140 BEEP .2,10
150 GOSUB VAL A$*100+100
160 GOTO 20
200 GOSUB 1000
210 FOR K=7 TO 23 STEP 2
220 PRINT AT 10,K;“*”
230 LET R$=S$((K-5)/2)
240 GOSUB 1100
250 IF C=0 THEN GOTO 240
260 PRINT AT 10,K;“□”
270 NEXT K
280 CLS : GOTO 1300
300 GOSUB 1000
310 FOR K=1 TO 20
320 LET RN=INT (RND*9)*2+1
330 PRINT AT 10,RN+6;“*”: LET
   R$=S$((RN+1)/2)
340 GOSUB 1100: IF C=0 THEN GOTO 340
350 PRINT AT 10,RN+6;“□”
360 NEXT K
370 CLS : GOTO 1300
400 CLS : PRINT “TYPE THE LETTER SHOWN
   ON THE□□□□SCREEN”
410 FOR N=1 TO 100: NEXT N
420 POKE 23672,0: POKE 23673,0
430 FOR K=1 TO 20
440 LET RN=INT (RND*9)+1
450 PRINT AT 11,16;S$(RN)
460 LET R$=S$(RN)
470 GOSUB 1100: IF C=0 THEN GOTO 470
480 PRINT INVERSE 1;AT 11,16;“□”: NEXT K
490 CLS : GOTO 1300
```

```
500 CLS : PRINT "TYPE THE WORD
   SHOWN ON THE □□□□□□
   SCREEN": POKE 23672,Ø: POKE 23673,Ø
510 LET TL=Ø: FOR N=1 TO 2Ø: RESTORE :
   LET RN=INT (RND*24)+1
520 FOR K=1 TO RN: READ T$: NEXT K
530 PRINT AT 1Ø,13;"□□□□□": PRINT
   AT 1Ø,13;T$
540 FOR M=1 TO LEN T$: PRINT AT
   9,11+M;"□*□□□□"
550 LET R$=T$(M): GOSUB 11ØØ
560 IF C=Ø THEN GOTO 55Ø
570 NEXT M
580 LET TL=TL+LEN T$: NEXT N
590 CLS : PRINT AT 18,Ø;"WORDS
   PER MINUTE=□";INT (TL*5ØØ/
   (PEEK 23672+256*PEEK 23673)+.5):
   GOTO 13ØØ
600 CLS : PRINT "TYPE THE WORDS
   SHOWN ON THE□□□□□□
   SCREEN": LET T$=""
610 FOR N=1 TO 6: RESTORE : LET
   RN=INT (RND*24)+1: FOR K=1 TO RN:
   READ X$: NEXT K
620 LET T$=T$+X$+"□"
630 NEXT N: LET T$=T$ (TO LEN T$-1)
640 POKE 23672,Ø: POKE 23673,Ø: PRINT AT
   1Ø,Ø;T$
650 PRINT AT 12,Ø;: FOR M=1 TO LEN T$
660 LET R$=T$(M): GOSUB 11ØØ
670 IF C=Ø THEN GOTO 66Ø
680 PRINT T$(M);: NEXT M
690 LET TL=LEN T$: GOTO 59Ø
1000 CLS : PRINT "PRESS THE KEY
   INDICATED BY THE□□ASTERISK"
1010 PRINT AT 12,7;"A□S□D□F□G□
   H□J□K□L"
1020 FOR K=1 TO 1ØØ: NEXT K: POKE
   23672,Ø: POKE 23673,Ø
1030 RETURN
1100 PAUSE Ø
1110 LET A$=INKEY$: IF A$< >R$ THEN
   BEEP .2,-1Ø: LET ER=ER+1: LET
   C=Ø: RETURN
1120 BEEP .Ø5,2Ø: LET C=1: RETURN
1300 PRINT AT 19,Ø;"TIME=□";
   (PEEK 23672+256*PEEK 23673)/5Ø;
   "□SECONDS": PRINT "NUMBER OF
   ERRORS=□";ER
1310 PAUSE 1ØØ: RETURN
2000 DATA "ASK","SAD","DAD","GLAD",
   "HAD","LASH","LAG","HAS","HASH",
   "DASH","ASS","ASH"
2010 DATA "ALL","FALL","HAG",
   "GASH","FLAG","JAG","LASS","FAG",
   "GAS","SASH","FLASK","SLASH"
```

```
10 DIM W$(21),WO$(33):FORZ=1TO33:
   READ WO$(Z):NEXT
20 FOR Z=1TO4Ø:LI$=LI$+"—":
   NEXT:LI$="▙"+LI$
30 PRINT "♥π"TAB(7)"*******TYP♠ING
   πTUT♠OR*****"
40 A$="ASDFGHJKL:":POKE 54296,15:
   GOTO 38Ø
50 TI$="ØØØØØØ":S=Ø:WW=1
60 N=Ø:POKE 198,Ø
```

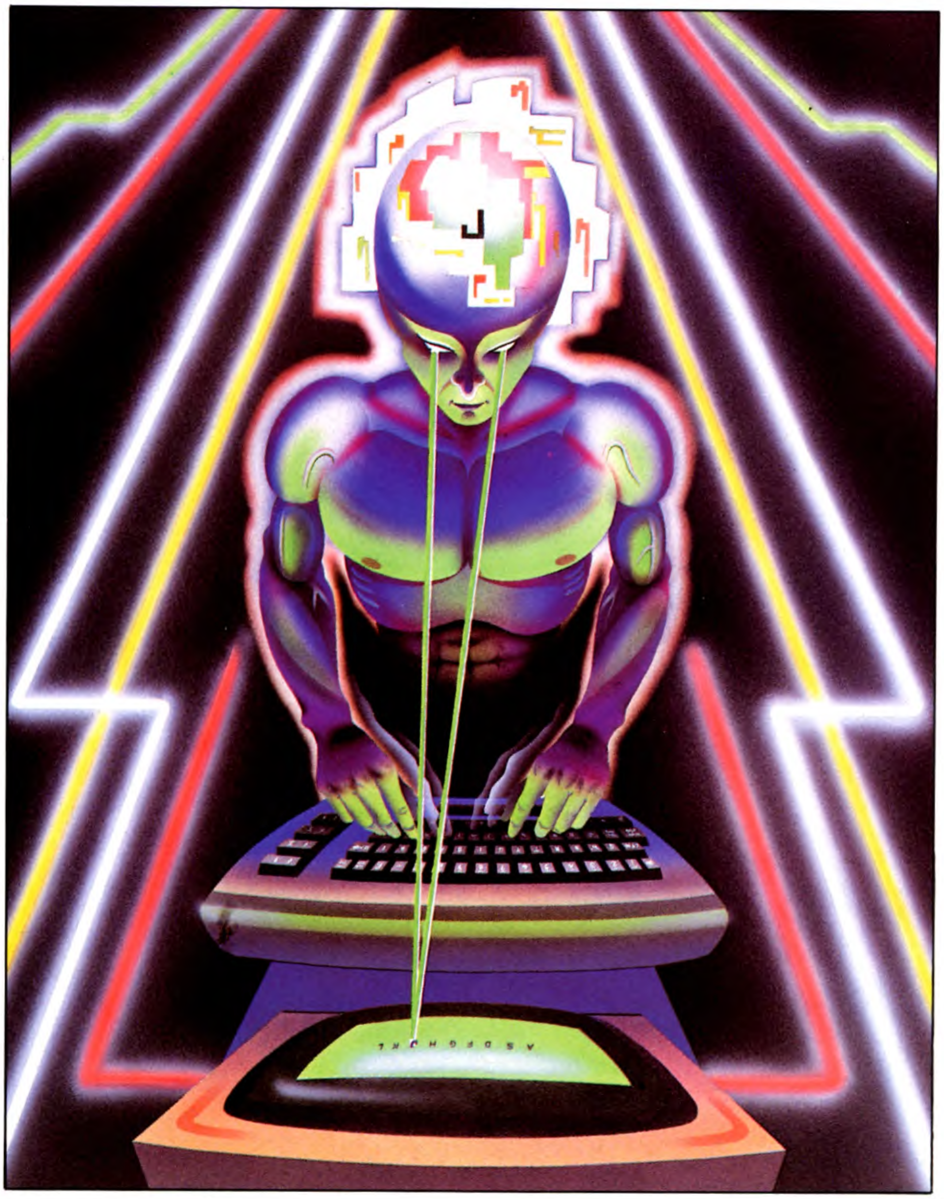


```
70 IF K=5 AND P>Ø THEN N=N+1:
   P=P+1:GOTO 14Ø
80 PRINT "▤▦▦▦▦▦";:IF K<3
   THEN PRINT TAB(11)"◣▨A□S□
   D□F□G□H□J□K□L□:":
   POKE 198,Ø
90 X=INT(RND(1)*1Ø)+1:N=N+1:
   P=P+1:IF K=1 THEN X=N: GOTO 12Ø
1ØØ IFK=3THENX=INT(RND(1)*1Ø)
   +1:PRINTTAB(18)"◣◿⊟◺▦▌▌
   ▌▌▌▌◫✚"MID$(A$,X,1)"◣◫▦▌▌
   ▌▌▌▌◹⊟◸"
11Ø IF K=4 THEN PRINT TAB(16)
   "◣□□□□□▌▌▌▌▌▌▌▌▌▌"
   W$(WW):X=N*.5+3
12Ø IF K<3 OR K=4 THEN PRINT
   "▤▦▦▦▦"TAB(11+(X-1)
   *2)"☰*"
13Ø IF K=5 THEN FOR Z=1 TO 7:
   PRINT "π"W$(Z);:NEXT Z:PRINT:
   PRINT "▨◣";
14Ø GET K$:IF K$="" THEN 14Ø
15Ø IF K>3 AND K$=MID$(W$(WW),N,1)
   THEN 21Ø
16Ø IF K>3 THEN 18Ø
17Ø IF K$=MID$(A$,X,1) THEN 21Ø
18Ø W=54276:A=54277
19Ø POKE W,33:POKE A,5Ø:POKE
   54273,3Ø:S=S+1
2ØØ POKE 54273,Ø:POKE W,32:GOTO 14Ø
21Ø W=54276:A=54277
22Ø IF K<3 OR K=4 THEN PRINT
   TAB(11+(X-1)*2)"○□"
23Ø IF K=5 THEN PRINT K$;
24Ø POKE W,33:POKE A,5Ø:POKE 54273,15Ø
25Ø POKE 54273,Ø:POKE W,32
26Ø IF K=1 AND N=1Ø THEN 29Ø
27Ø IF K>3 AND N<LEN(W$(WW)) THEN
   7Ø
28Ø IF K<4 AND N<2Ø THEN 7Ø
29Ø IF K>3 THEN NEXT WW
3ØØ WW=VAL(MID$(TI$,3,2))*6Ø
   +VAL(RIGHT$(TI$,2))
31Ø PRINT "☰▽"LI$;:PRINT "πYOUR
   TIME WAS▓"WW"SEC";
32Ø PRINT "▙▙πMISTAKES MADE:▓"S
33Ø IF K>3 THEN PRINT "▦"TAB(7)
   INT(1Ø*NU/WW)"☰WORDS PER
   MINUTE□"
34Ø PRINT LI$:GOTO 53Ø
35Ø NU=Ø:P=Ø:S=Ø:FORWW=1
   TOMM:W$(WW)=WO$(INT(RND(1)
   *33)+1)
36Ø IF K=5 AND WW<>7 THEN
   W$(WW)=W$(WW)+"□"
37Ø NU=NU+LEN(W$(WW)):NEXT WW:
   TI$="ØØØØØØ":FOR WW=1 TO MM:
   GOTO 6Ø
38Ø POKE 5328Ø,6:POKE 53281,Ø:
   POKE 198,Ø
39Ø PRINT "▤▦▦▦▦▦▦▦▦
   ▦▦☰"TAB(15)"OPTIONS▦✚"
4ØØ FOR Z=1 TO 5:PRINT TAB(13)Z;
   ":TEST";Z:NEXT Z
41Ø PRINT TAB(12)"☰▦ENTER CHOICE?"
42Ø GET K$:K=VAL(K$):IF K<1 OR K>5
   THEN 42Ø
43Ø PRINT"▽"LI$"✚"
44Ø IF K<3 THEN PRINT "○PRESS THE KEY
   INDICATED BY THE ASTERISK"
45Ø IF K=3 THEN PRINT "○□□TYPE THE
   LETTER SHOWN ON THE SCREEN"
46Ø IF K=4 THEN PRINT"○TYPE THE
   WORD THAT APPEARS ON THE
   SCREEN";:MM=2Ø
47Ø IF K=5 THEN PRINT TAB(12)"○TYPE
   THESE WORDS":MM=7
48Ø PRINT LI$:FOR Z=1 TO 1ØØØ:NEXT
49Ø PRINTTAB(11)"☰PRESS KEY TO
   START":POKE 198,Ø:WAIT 198,1
5ØØ PRINTTAB(11)"○□□□□□□□□
   □□□□□□□□□□□□□"
51Ø ON K GOTO 5Ø,5Ø,5Ø,35Ø,35Ø
52Ø GOTO 38Ø
53Ø FOR Z=1 TO 1ØØØ:NEXT Z: GOTO 38Ø
54Ø DATA ASK,SAD,DAD,GLAD,HAD,
   FLASH,FLASK,LASH,SLASH,LAG,HAS,
   HAJ,SKA,HAS,HASH
55Ø DATA DASH,ASS,SHALL,ASH,ALL,
   FALL,HAG,GASH,FLAG,JAG,GAL,LASS,
   FAG,GAS,GAFF
56Ø DATA JAH,SASH,AS
```

```
1Ø MODE1
2Ø DIM A$(32) :VDU23;82Ø2;Ø;Ø;Ø;
3Ø FOR T=1 TO 32:READ A$(T):NEXT
4Ø A$="ASDFGHJKL;":A2$=
   "A□S□D□F□G□H□J□K□L□;"
5Ø PRINTTAB(5,15)"WHICH LEVEL DO YOU
   WANT (1-5)";
6Ø G=GET-48
7Ø IF G<1 OR G>5 THEN 6Ø
8Ø ON G GOTO 9Ø,12Ø,15Ø,18Ø,21Ø
9Ø PROCLEV1
1ØØ PROCANGO
11Ø GOTO 8Ø
12Ø PROCLEV2
13Ø PROCANGO
14Ø GOTO 8Ø
15Ø PROCLEV3
16Ø PROCANGO
17Ø GOTO 8Ø
18Ø PROCLEV4
19Ø PROCANGO
2ØØ GOTO 8Ø
21Ø PROCLEV5
22Ø PROCANGO
23Ø GOTO 8Ø
24Ø DEF PROCANGO
25Ø CLS
26Ø COLOUR3
27Ø PRINTTAB(1Ø,4)"LEVEL NUMBER□";TN
28Ø PRINTTAB(1Ø,6)"TOOK YOU□";
   TM/1ØØ;"□SECONDS"
29Ø IF B=Ø THEN B$="NO" ELSE
   B$=STR$(B)
3ØØ PRINTTAB(1Ø,8)"AND YOU
   HAD□";B$;"□ERRORS";
31Ø IF TN=4 OR TN=5 THENPRINT
   TAB(1Ø,1Ø);WM;"□WORDS PER MINUTE"
33Ø PRINTTAB(5,15)"WHICH LEVEL DO YOU
   WANT (1-5)";
34Ø G=GET-48
35Ø IF G<1 OR G>5 THEN 34Ø
36Ø ENDPROC
37Ø DEF PROCLEV1
38Ø CLS
39Ø PRINT""PRESS THE KEY INDICATED BY
   THE ASTERISK"
4ØØ B=Ø:TN=1
41Ø COLOUR1
42Ø PRINTTAB(1Ø,1Ø)A2$
43Ø FOR T=1 TO 1Ø
44Ø B2$=MID$(A$,T,1)
45Ø COLOUR3
46Ø PRINTTAB(8+T*2,9)"*"TAB
   (8+T*2,1Ø)B2$
47Ø B$=GET$:IF T=1 THEN TIME=Ø
48Ø IF B$=B2$ THEN 52Ø
49Ø SOUND1,-1Ø,1,1
5ØØ B=B+1
51Ø GOTO 47Ø
52Ø COLOUR1
53Ø PRINTTAB(8+T*2,9)"□"TAB
   (8+T*2,1Ø)B2$
54Ø SOUND1,-1Ø,2ØØ,1
55Ø NEXT
56Ø TM=TIME
57Ø ENDPROC
58Ø DEF PROCLEV2
59Ø CLS
6ØØ PRINT""PRESS THE KEY INDICATED BY
   THE ASTERISK"
61Ø B=Ø:TN=2
62Ø COLOUR1
63Ø PRINTTAB(1Ø,1Ø)A2$
64Ø FOR T=1 TO 2Ø
65Ø P=RND(1Ø):X=P*2+8
66Ø B2$=MID$(A$,P,1)
67Ø COLOUR3
68Ø PRINTTAB(X,9)"*"TAB(X,1Ø)B2$
69Ø B$=GET$:IF T=1 THEN TIME=Ø
7ØØ IF B$<>B2$ THEN SOUND1,1Ø,1,1:
   B=B+1:GOTO 69Ø
71Ø SOUND1,-1Ø,2ØØ,1
72Ø COLOUR1
73Ø PRINTTAB(X,9)"□"TAB(X,1Ø)B2$
74Ø NEXT
75Ø TM=TIME
76Ø ENDPROC
77Ø DEF PROCLEV3
78Ø B=Ø:TN=3
```

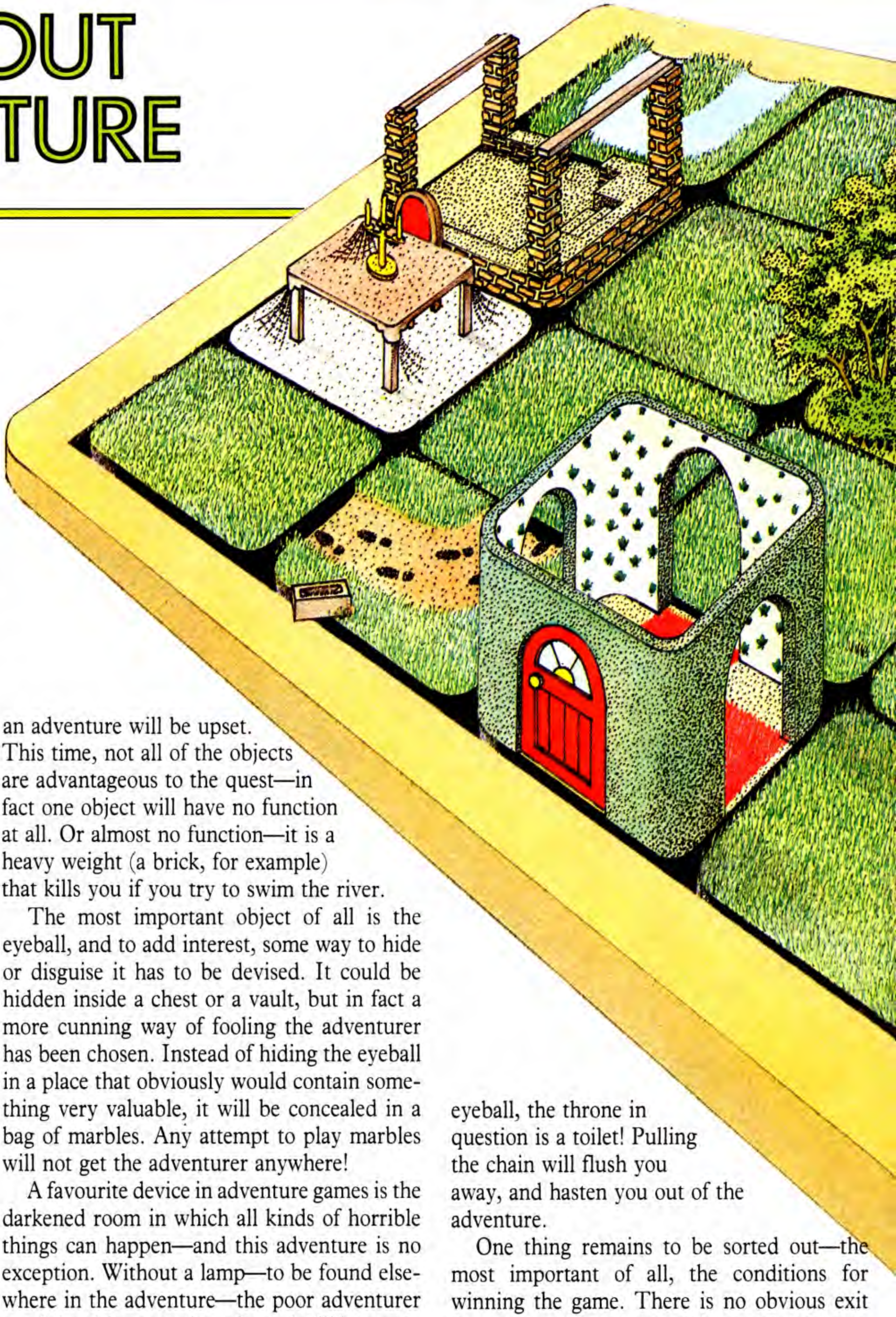
```
790 CLS
800 PRINT''“PRESS THE KEY SHOWN”
810 FOR T=1 TO 2Ø
820 P=RND(1Ø):B2$=MID$(A$,P,1)
830 PRINTTAB(19,15)B2$
840 B$=GET$:IF T=1 THEN TIME=Ø
850 IF B$=B2$ THEN SOUND1,-1Ø,
   2ØØ,1 ELSE SOUND1,-1Ø,1,1:
   B=B+1:GOTO 84Ø
860 NEXT
870 TM=TIME
880 ENDPROC
890 DEF PROCLEV4
900 B=Ø:TN=4
910 CLS
920 PRINT''“TYPE WORD SHOWN ON THE
   SCREEN”
930 NC=Ø
940 FOR T=1 TO 2Ø
950 P=RND(32)
960 B2$=A$(P)
970 NC=NC+LEN(B2$)
980 PRINTTAB(17,14) B2$“□□□”
990 B$=“”
1000 FOR X=1 TO LEN(B2$)
1010 PRINTTAB(16+X,13)“*”
1020 B$=MID$(B2$,X,1):B3$=GET$:
   IF T=1 AND X=1 THEN TIME=Ø
1030 IF B$=B3$ THEN SOUND1,-1Ø,
   2ØØ,1 ELSE SOUND1,-1Ø,1,1:
   B=B+1:GOTO 1Ø2Ø
1040 PRINTTAB(16+X,13)“□”
1050 NEXT
1060 NEXT
1070 TM=TIME
1080 WM=INT(NC*1ØØØ/TM+.5)
1090 ENDPROC
1100 DEF PROCLEV5
1110 B=Ø:TN=5
1120 NC=Ø
1130 CLS
1140 PRINT''“TYPE THE WORDS IN
   CORRECTLY”
1150 B2$=A$(RND(32))
1160 NC=LEN(B2$)
1170 FOR T=1 TO 6
1180 B2$=B2$+“□”+A$(RND(32))
1190 NEXT
1200 PRINTTAB(Ø,14)B2$
1210 PRINT
1220 NC=LEN(B2$)
1230 FOR X=1 TO LEN(B2$)
1240 B$=GET$:IF X=1 THEN TIME=Ø
1250 B3$=MID$(B2$,X,1)
1260 IF B$=B3$ THEN SOUND1,-1Ø,
   2ØØ,1 ELSE SOUND1,-1Ø,1,1:
   B=B+1:GOTO 124Ø
1270 PRINTB$;
1280 NEXT
1290 PRINT
1300 TM=TIME
1310 WM=INT(NC*1ØØØ/TM+.5)
1320 ENDPROC
1330 DATA ASK,SAD,DAD,GLAD,HAD,
   FLASH,FLASK,LASH,SLASH,LAG,HAS,
   HAJ,JAFFA,SKA,HASH,DASH,ASS,
   SHALL, ASH,ALL,FALL,HAG,GASH,
   FLAG,JAG, GAL,LASS,FAG,GAS,AS,
   JAH,SASH
```

```
10 OB$=“A□S□D□F□G□H□J□
   K□L□;”
20 CLS
30 DIM W$(28)
40 FOR K=1 TO 28
50 READ W$(K)
60 NEXT
70 PRINT@71,“WHICH LEVEL OF”
80 PRINT@1Ø1,“DIFFICULTY (1-5) ?”
90 PRINT@165,“TYPE (Ø) TO QUIT”
100 A$=INKEY$:IF A$<“Ø” OR A$>“5”
   THEN 1ØØ
110 A=VAL(A$):ON A+1 GOSUB 999,
```



```
   200,300,400,600,800
120 ER=Ø
130 GOTO 7Ø
200 GOSUB1ØØØ
21Ø AP=1252
22Ø FOR K=1 TO 1Ø
23Ø AP=AP+2
24Ø GOSUB 11ØØ
25Ø NEXT
26Ø CLS:PRINT@448,“TIME =”;
   TIMER/5Ø;“SECONDS”
27Ø PRINT“NUMBER OF ERRORS
   =”;ER;
28Ø RETURN
3ØØ GOSUB 1ØØØ
31Ø FOR K=1 TO 2Ø
32Ø AP=1252+2*RND(1Ø)
33Ø GOSUB 11ØØ
34Ø NEXT
35Ø CLS
36Ø PRINT@448,“TIME =”;
   TIMER/5Ø;“SECONDS”
37Ø PRINT“NUMBER OF ERRORS
   =”;ER;
38Ø RETURN
4ØØ CLSØ:PRINT“□TYPE THE LETTER
   SHOWN ON THE□□□□□□□□
   SCREEN”
41Ø PRINT@2Ø6,“□□□”;:PRINT@238,
   “□□□”;:PRINT@27Ø,“□□□”;
42Ø FOR K=1 TO 2Ø
43Ø P$=MID$(OB$,2*RND(1Ø)−1,1)
44Ø PRINT@239,P$;
45Ø A$=INKEY$:IF A$=“” THEN 45Ø
46Ø IF K=1 THEN TIMER=Ø
47Ø IF A$< >P$ THEN SOUND 5,1:
   ER=ER+1:GOTO 45Ø
48Ø POKE 1263,128
49Ø SOUND 2ØØ,1
5ØØ NEXT
51Ø GOTO 35Ø
6ØØ CLS:PRINT“□TYPE THE WORD THAT
   APPEARS”
61Ø T=Ø
62Ø FOR K=1 TO 1Ø
63Ø P$=W$(RND(28))
64Ø T=T+LEN(P$)
65Ø PRINT@237,P$
66Ø P=1
67Ø POKE 1228+P,1Ø6
68Ø A$=INKEY$:IF A$=“” THEN 68Ø
69Ø IF K=1 THEN TIMER=Ø
7ØØ IF A$< >MID$(P$,P,1) THEN
   ER=ER+1:SOUND 5,1:GOTO 67Ø
71Ø POKE 1228+P,96
72Ø SOUND 2ØØ,1:P=P+1:IF
   P< =LEN(P$) THEN 67Ø
73Ø NEXT
74Ø CLS
75Ø PRINT@416, USING“WORDS PER
   MINUTE = ###.##”;T*5ØØ/TIMER
76Ø GOTO 36Ø
8ØØ CLS:P$=“”:FOR K=1 TO 6
81Ø P$=P$+W$(RND(28))+“□”
82Ø NEXT
83Ø P$=LEFT$(P$,LEN(P$)−1)
84Ø PRINT“□TYPE THESE WORDS”
85Ø P=1:PRINT@224,P$
86Ø A$=INKEY$:IF A$=“” THEN 86Ø
87Ø TIMER=Ø:GOTO 89Ø
88Ø A$=INKEY$:IF A$=“” THEN 88Ø
89Ø IF A$< >MID$(P$,P,1) THEN
   ER=ER+1:SOUND 5,1:GOTO 88Ø
9ØØ POKE 1311+P,PEEK(1247+P)
91Ø SOUND 2ØØ,1:P=P+1:IF
   P< =LEN(P$) THEN 88Ø
92Ø T=LEN(P$):GOTO 74Ø
999 CLS:END
1ØØØ CLS:PRINT“PRESS THE KEY
   INDICATED BY THE□□□□□
   ASTERISK”
1Ø1Ø PRINT@262,OB$
1Ø2Ø FOR K=1 TO 1ØØØ:NEXT
1Ø3Ø RETURN
11ØØ POKE AP,1Ø6
111Ø A$=INKEY$:IF A$=“” THEN 111Ø
112Ø IF K=1 THEN TIMER=Ø
113Ø IF (ASC(A$)OR64)< >PEEK
   (AP+32) THEN SOUND 5,1:ER=
   ER+1:GOTO 111Ø
114Ø SOUND 2ØØ,1
115Ø POKE AP,96
116Ø RETURN
9ØØØ DATA ASK,SAD,DAD,GLAD,HAD,
   LASH,LAG,HAS,HASH,DASH,ASS,ASH,
   HAG,ALL,JAG,FAG,GAS
9Ø1Ø DATA HALF,FALL,GASH,FLAG,
   LASS,SASH,FLASK,SLASH,FLASH,
   SHALL,JAFFA
```

MAPPING OUT AN ADVENTURE



The very first stage in writing an adventure is to work out the general story line and draw a rough map of all the locations. This then forms the basis of the whole program

It is essential to work out the whole story before you start programming. If you don't, you're likely to have lots of difficulty, with many bugs and loose ends to tie up.

To see how it's done, the next few articles in *INPUT*'s Games Programming course follow the development of a typical (although necessarily very short) adventure program. The idea for the *INPUT* adventure is set in some faraway land where the player has to recover the fabled lost eyeball of the purple icon. If you follow all the steps in writing this adventure you'll quickly see how to write your own.

THE STORY

You have to design a world that will fit in with the broad storyline. Suitable objects, and roles for them have to be found and puzzles have to be devised.

You don't need to work all this out at once, because as the adventure is thought about, more and more of the story becomes worked out and the details start to slot into place. So start by roughing out the storyline.

The player in the *INPUT* adventure is in dire financial difficulty, and has set out on a quest to find the fabulous (and very valuable) eyeball that is hidden somewhere in the world of the adventure. Unfortunately the Inland Revenue has sent a tax inspector in pursuit. The tax inspector's role in the adventure is somewhat like the pirate's role in many other adventures—namely, to wreak havoc on the poor adventurer. If the tax inspector does appear, one of two things can happen. If you are carrying an object, he will take it to offset your huge tax bill, but if you haven't yet found anything (and therefore can't pay), he locks you in a deep dungeon.

So much for the bare bones of the idea. Now you should fill in some of the details—like what objects are to be found on the quest. In the *INPUT* adventure, we have decided that the usual rule of collecting all the objects in an adventure will be upset. This time, not all of the objects are advantageous to the quest—in fact one object will have no function at all. Or almost no function—it is a heavy weight (a brick, for example) that kills you if you try to swim the river.

The most important object of all is the eyeball, and to add interest, some way to hide or disguise it has to be devised. It could be hidden inside a chest or a vault, but in fact a more cunning way of fooling the adventurer has been chosen. Instead of hiding the eyeball in a place that obviously would contain something very valuable, it will be concealed in a bag of marbles. Any attempt to play marbles will not get the adventurer anywhere!

A favourite device in adventure games is the darkened room in which all kinds of horrible things can happen—and this adventure is no exception. Without a lamp—to be found elsewhere in the adventure—the poor adventurer won't be able to see the exits and will be stuck. This may be a little unfair because no warning of impending doom is given, and there is no way for the player to get out of the darkened room unless he has *already* found the lamp.

To counteract the attentions of the tax inspector, and give the adventurer a better chance, a weapon will be hidden somewhere in the adventure—either a gun or a knife perhaps.

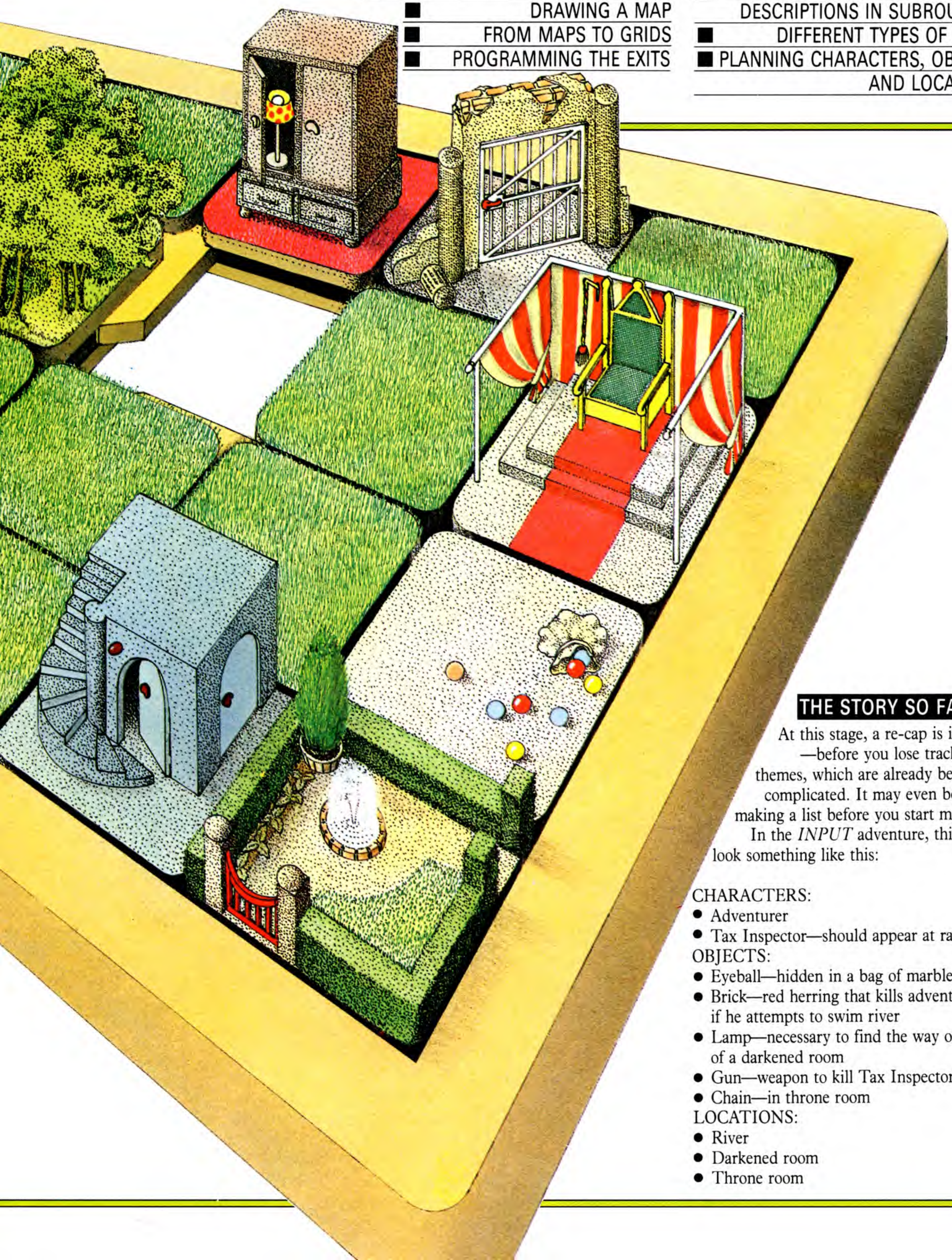
Finally, just for fun, there is a throne room and a chain. The throne room is not quite what it seems. In fact, if you are not carrying the eyeball, the throne in question is a toilet! Pulling the chain will flush you away, and hasten you out of the adventure.

One thing remains to be sorted out—the most important of all, the conditions for winning the game. There is no obvious exit from the world, and part of the puzzle is how to escape with the eyeball.

To win the game, the adventurer must obviously have found the eyeball—not just the bag of marbles. To make it more difficult, the adventurer must also be in the throne room. Pulling the chain this time will not flush the adventurer down the loo!

The advantage of making the way out also a hazard under different conditions, is that it is likely to discourage the wary player from trying it too often, thus prolonging the play.

- THE STORY LINE
- DRAWING A MAP
- FROM MAPS TO GRIDS
- PROGRAMMING THE EXITS
- ENTERING THE LOCATION DESCRIPTIONS IN SUBROUTINES
- DIFFERENT TYPES OF GRIDS
- PLANNING CHARACTERS, OBJECTS AND LOCATIONS



THE STORY SO FAR . . .

At this stage, a re-cap is in order—before you lose track of the themes, which are already becoming complicated. It may even be worth making a list before you start mapping. In the *INPUT* adventure, this might look something like this:

CHARACTERS:
- Adventurer
- Tax Inspector—should appear at random

OBJECTS:
- Eyeball—hidden in a bag of marbles
- Brick—red herring that kills adventurer if he attempts to swim river
- Lamp—necessary to find the way out of a darkened room
- Gun—weapon to kill Tax Inspector
- Chain—in throne room

LOCATIONS:
- River
- Darkened room
- Throne room

So far, the adventure has only fixed three of the locations because of things which need to take place there. You might well have decided upon more by this stage. But whatever the case, your next step is to fit all these themes together in a map of the adventure world.

STARTING MAPPING

Your first map will probably consist simply of a series of boxes, connected by arrows, rather like **fig. 1.** Each of the boxes represents a room or location in the world—location is probably the better term because you are not confined indoors with your adventures, and the location might be anything from on top of a pinhead in the queen's hem to a large plain, stretching as far as the eye can see. You will need to incorporate all the locations in your preliminary list, plus others to tie the game together.

When you are drawing this map remember to mark in the directions that you can go from each room, because you may wish to have exits which only work in one direction—accompanied by a message such as

THE DOOR SLAMS SHUT BEHIND YOU

The dotted lines leading from the Dark Room indicate that the adventurer will only be able to go in that direction if certain conditions are satisfied. In this case, the condition is that the adventurer has the lamp and has lit it so that he can see the exits.

It is very difficult to predict quite how many locations you will be able to fit into a given amount of RAM. The difficulty arises because there are so many things competing for memory space in an adventure program—room descriptions, the words you wish the program to recognise, the number of objects and what you want to do with them, the number of puzzles and their complexity, and so on.

Once you have written a few small adventures and checked how much memory is occupied—see page 268—you'll have some idea of what will fit in your machine.

Owners of 16K computers will soon discover that it is almost impossible to write a large-scale adventure in such a small amount of RAM. However the adventure which you will see developed over the next few issues of *INPUT* has only a small number of rooms—12 in fact—and therefore is too small to cause any headaches.

A map for the Quest for the Jewelled Eyeball would look like that in **fig. 1.** The location descriptions should be kept short at this stage. The links have been drawn in, and the starting point has been decided. This is quite important as it affects the way the adventure is tackled, and the order in which the objects are found and the puzzles attempted.

The objects are also marked in their locations. The objects that will appear later on, for example, the eyeball, are best listed at the side of the map.

FROM MAPS TO GRIDS

Once the map has been completed, the information can be transferred on to a grid.

There are two forms of grid commonly used when planning adventure games: one based on squares as in **fig. 2**, and one based on octagons as in **fig. 3.** Which type you use will depend on the number of possible exits from each of the locations.

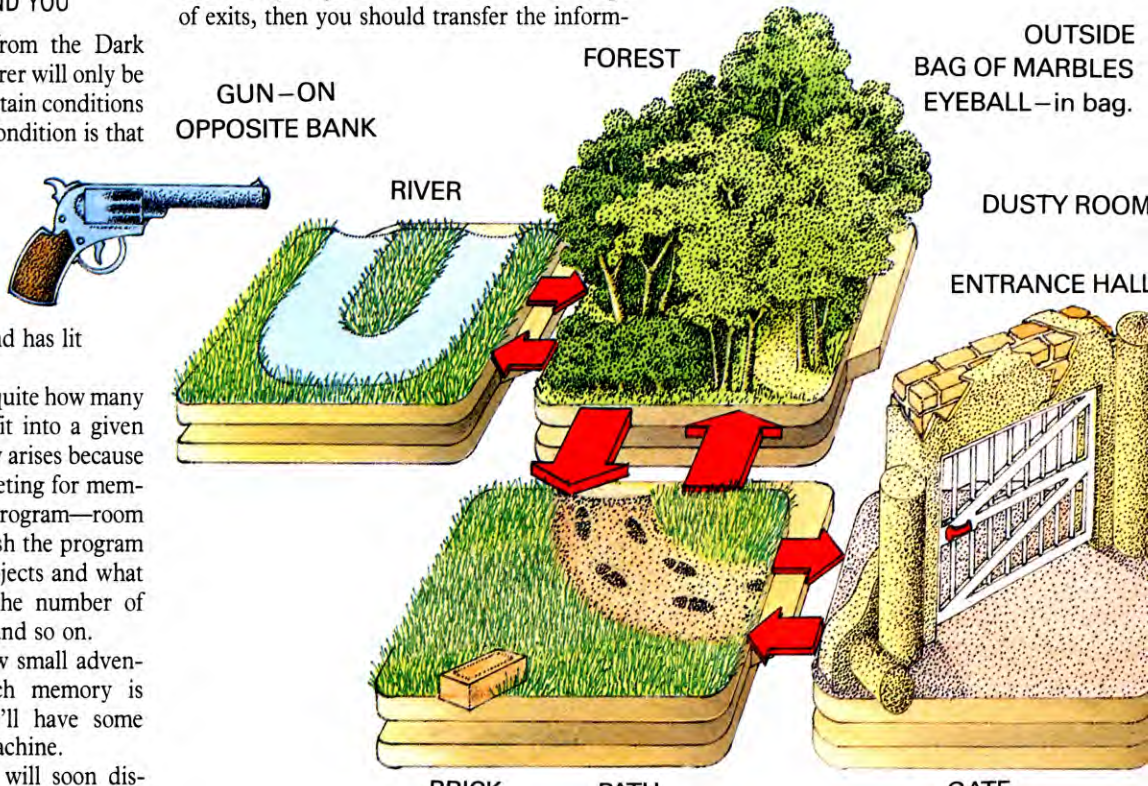
The simplest kind of adventures only call for you to exit from rooms in four directions: North, South, East and West (as in the eyeball adventure). If your adventure uses this range of exits, then you should transfer the information to the square grid so that it matches the conditions on your map. How this is done is developed in detail below. If you have used exits including Northeast, Northwest, Southeast and Southwest, you should use the octagonal grid, but this kind of grid system is very complicated.

The final variation on the grid is if you have chosen to move up and down as well. In this case the best solution is to use a separate grid for each 'level' of the adventure.

The eyeball adventure is based on the square type of grid—allowing exits to the North, South, East and West only. Unless there is a real need for other directions this kind of adventure is quite adequate, and there is a way of 'fiddling' up and down directions. If you use a description including a phrase like:

THERE IS A STAIRCASE DESCENDING TO THE WEST

you can use the normal West response and routine instead.



1. The first map of the adventure shows all the planned locations in their relative positions. The solid arrows indicate exits which are open at all times, but the striped arrows indicate exits which can only be used under special conditions—in this case when the lamp has been lit

THE GRID

This adventure will need a grid of six squares by four squares—check this by counting the maximum number of locations on your map, up and down and right to left. Before you start transferring all the details to the grid, make sure that you have numbered each of the squares. Start at number 1 in the top left, and work through to the bottom right.

Once the squares have been numbered and the room details transferred, the grid appears as in **fig. 4** (overleaf).

STARTING THE PROGRAM

Now that you have a story line and a completed grid you can start on the program.

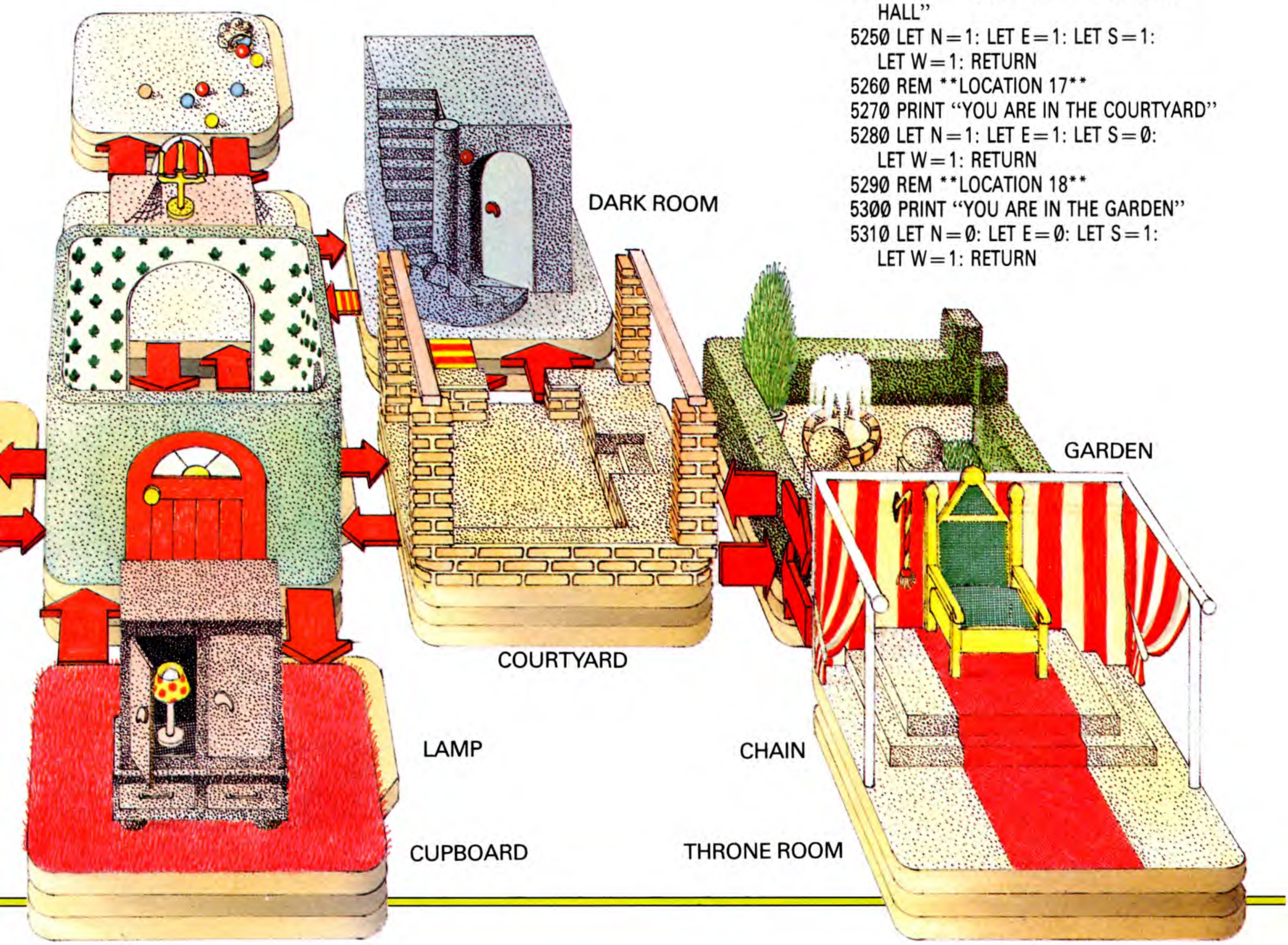
The first stage is to type in the location descriptions, based on your grid. You must decide how long these are going to be. Aim to convey as much of the atmosphere of the adventure as possible without wasting memory.

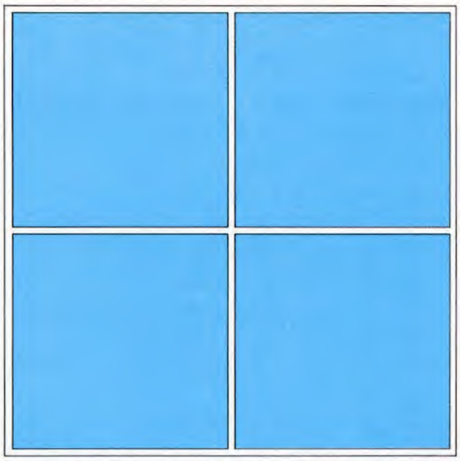
Along with the location descriptions the computer must be told in which directions the exits are in.

At last, here are the first sections of program. The high line numbers are to allow plenty of space for earlier program sections as you develop the game.

Type your section in and SAVE it on tape:

```
5000 REM **LOCATION DESCRIPTION**
5010 REM **LOCATION 4**
5020 PRINT "YOU ARE OUTSIDE A LARGE
   BUILDING"
5030 LET N=0: LET E=0: LET S=1:
   LET W=0: RETURN
5040 REM **LOCATION 7**
5050 PRINT "YOU ARE BY A FAST-
   FLOWING RIVER"
5060 LET N=0: LET E=1: LET S=0:
   LET W=0: RETURN
5070 REM **LOCATION 8**
5080 PRINT "YOU ARE IN A PETRIFIED
   FOREST"
5090 LET N=0: LET E=0: LET S=1:
   LET W=1: RETURN
5100 REM **LOCATION 10**
5110 PRINT "YOU ARE IN A DUSTY ROOM"
5120 LET N=1: LET E=1: LET S=1:
   LET W=0: RETURN
5130 REM **LOCATION 11**
5140 PRINT "YOU ARE IN A DARK ROOM"
5150 IF LA<>1 THEN LET N=0: LET
   E=0: LET S=0: LET W=0:
   PRINT "IT IS TOO DARK TO SEE THE
   EXITS": LET DA=1: RETURN
5160 LET N=0: LET E=0: LET S=1:
   LET W=1: RETURN
5170 REM **LOCATION 14**
5180 PRINT "YOU ARE ON A MUDDY PATH"
5190 LET N=1: LET E=1: LET S=0:
   LET W=0: RETURN
5200 REM **LOCATION 15**
5210 PRINT "YOU ARE BY THE GATE TO
   THE□□□□□□HIDDEN CITY"
5220 LET N=0: LET E=1: LET S=0:
   LET W=1: RETURN
5230 REM **LOCATION 16**
5240 PRINT "YOU ARE IN THE ENTRANCE
   HALL"
5250 LET N=1: LET E=1: LET S=1:
   LET W=1: RETURN
5260 REM **LOCATION 17**
5270 PRINT "YOU ARE IN THE COURTYARD"
5280 LET N=1: LET E=1: LET S=0:
   LET W=1: RETURN
5290 REM **LOCATION 18**
5300 PRINT "YOU ARE IN THE GARDEN"
5310 LET N=0: LET E=0: LET S=1:
   LET W=1: RETURN
```



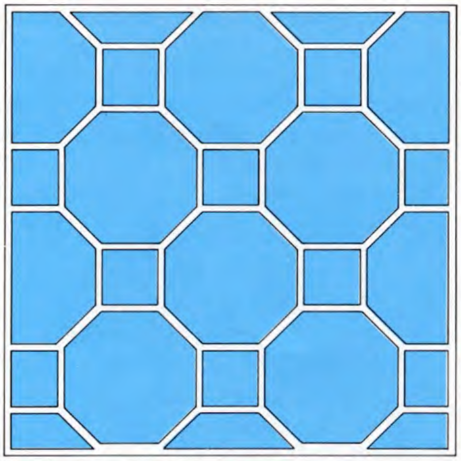


2. A small part of a square-based grid on which to plan adventures using four exits so that you can go to the north, south, east and west

```
5320 REM **LOCATION 22**
5330 PRINT “YOU ARE IN THE CUPBOARD”
5340 LET N=1: LET E=Ø: LET S=Ø:
   LET W=Ø: RETURN
5350 REM **LOCATION 24**
5360 PRINT “YOU ARE IN THE THRONE
   ROOM”
5370 LET N=1: LET E=Ø: LET S=Ø:
   LET W=Ø: RETURN
```

Vic 20 owners will have to add some spaces so that the PRINT statements format correctly on the Vic's 22 column screen.

```
5ØØØ REM ** LOCATION DESCRIPTION **
5Ø1Ø REM ** LOCATION 4 **
5Ø2Ø PRINT “YOU ARE OUTSIDE A LARGE
   BUILDING”
5Ø3Ø N=Ø:E=Ø:S=1:W=Ø:RETURN
5Ø4Ø REM ** LOCATION 7 **
5Ø5Ø PRINT “YOU ARE BY A FAST-
   FLOWING RIVER”
5Ø6Ø N=Ø:E=1:S=Ø:W=Ø:RETURN
5Ø7Ø REM ** LOCATION 8 **
5Ø8Ø PRINT “YOU ARE IN A PETRIFIED
   FOREST”
5Ø9Ø N=Ø:E=Ø:S=1:W=1:RETURN
51ØØ REM ** LOCATION 1Ø **
511Ø PRINT “YOU ARE IN A DUSTY ROOM”
512Ø N=1:E=1:S=1:W=Ø:RETURN
513Ø REM ** LOCATION 11 **
514Ø PRINT “YOU ARE IN A DARK ROOM”
515Ø IF OB(6)< >-1 OR LA< >1 THEN
   N=Ø:E=Ø:S=Ø:W=Ø
5155 IF OB(6)< >-1 OR LA< >1 THEN
   PRINT “IT'S TOO DARK TO SEE THE
   EXITS”:RETURN
516Ø N=Ø:E=Ø:S=1:W=1:RETURN
517Ø REM ** LOCATION 14 **
```



3. A small part of an octagonal grid—used when you want to include exits that go to the north west, north east, south west and south east

```
518Ø PRINT “YOU ARE ON A MUDDY PATH”
519Ø N=1:E=1:S=Ø:W=Ø:RETURN
52ØØ REM ** LOCATION 15 **
521Ø PRINT “YOU ARE BY THE GATE TO THE
   HIDDEN CITY.”
522Ø N=Ø:E=1:S=Ø:W=1:RETURN
523Ø REM ** LOCATION 16 **
524Ø PRINT “YOU ARE IN THE ENTRANCE
   HALL”
525Ø N=1:E=1:S=1:W=1:RETURN
526Ø REM ** LOCATION 17 **
527Ø PRINT “YOU ARE IN THE COURTYARD”
528Ø N=1:E=1:S=Ø:W=1:RETURN
529Ø REM ** LOCATION 18 **
53ØØ PRINT “YOU ARE IN THE GARDEN”
531Ø N=Ø:E=Ø:S=1:W=1:RETURN
532Ø REM ** LOCATION 22 **
533Ø PRINT “YOU ARE IN THE CUPBOARD”
534Ø N=1:E=Ø:S=Ø:W=Ø:RETURN
535Ø REM ** LOCATION 24 **
536Ø PRINT “YOU ARE IN THE THRONE
   ROOM”
537Ø N=1:E=Ø:S=Ø:W=Ø:RETURN
```

For the Acorn machines, the spaces in Line 521Ø should be omitted.

```
5ØØØ REM**LOCATION DESCRIPTION**
5Ø1Ø REM **LOCATION 4**
5Ø2Ø PRINT “YOU ARE OUTSIDE A LARGE
   BUILDING”
5Ø3Ø N=Ø:E=Ø:S=1:W=Ø:RETURN
5Ø4Ø REM **LOCATION 7**
5Ø5Ø PRINT “YOU ARE BY A FAST-
   FLOWING RIVER”
5Ø6Ø N=Ø:E=1:S=Ø:W=Ø:RETURN
5Ø7Ø REM **LOCATION 8**
5Ø8Ø PRINT “YOU ARE IN A PETRIFIED
   FOREST”
5Ø9Ø N=Ø:E=Ø:S=1:W=1:RETURN
51ØØ REM **LOCATION 1Ø**
511Ø PRINT “YOU ARE IN A DUSTY ROOM”
512Ø N=1:E=1:S=1:W=Ø:RETURN
513Ø REM **LOCATION 11**
514Ø PRINT “YOU ARE IN A DARK ROOM”
515Ø IF OB(6)< >-1 OR LA< >1 THEN
   N=Ø:E=Ø:S=Ø:W=Ø:PRINT“IT IS
   TOO DARK TO SEE THE EXITS ”: RETURN
516Ø N=Ø:E=Ø:S=1:W=1:RETURN
517Ø REM **LOCATION 14**
518Ø PRINT “YOU ARE ON A MUDDY PATH”
519Ø N=1:E=1:S=Ø:W=Ø:RETURN
52ØØ REM **LOCATION 15**
521Ø PRINT “YOU ARE BY THE GATE TO
   THE□□□□□□HIDDEN CITY”
522Ø N=Ø:E=1:S=Ø:W=1:RETURN
523Ø REM**LOCATION 16**
524Ø PRINT “YOU ARE IN THE ENTRANCE
   HALL”
525Ø N=1:E=1:S=1:W=1:RETURN
526Ø REM**LOCATION 17**
527Ø PRINT “YOU ARE IN THE COURTYARD”
528Ø N=1:E=1:S=Ø:W=1:RETURN
529Ø REM**LOCATION 18**
53ØØ PRINT “YOU ARE IN THE GARDEN”
531Ø N=Ø:E=Ø:S=1:W=1:RETURN
532Ø REM **LOCATION 22**
533Ø PRINT “YOU ARE IN THE CUPBOARD”
534Ø N=1:E=Ø:S=Ø:W=Ø:RETURN
535Ø REM **LOCATION 24**
536Ø PRINT “YOU ARE IN THE THRONE
   ROOM”
537Ø N=1:E=Ø:S=Ø:W=Ø:RETURN
```

Don't worry about the liberal use of memory-eating REM statements. At this early stage of program development it is very important to know what each piece of program does, or which location number a particular description refers to. They can always be removed later.

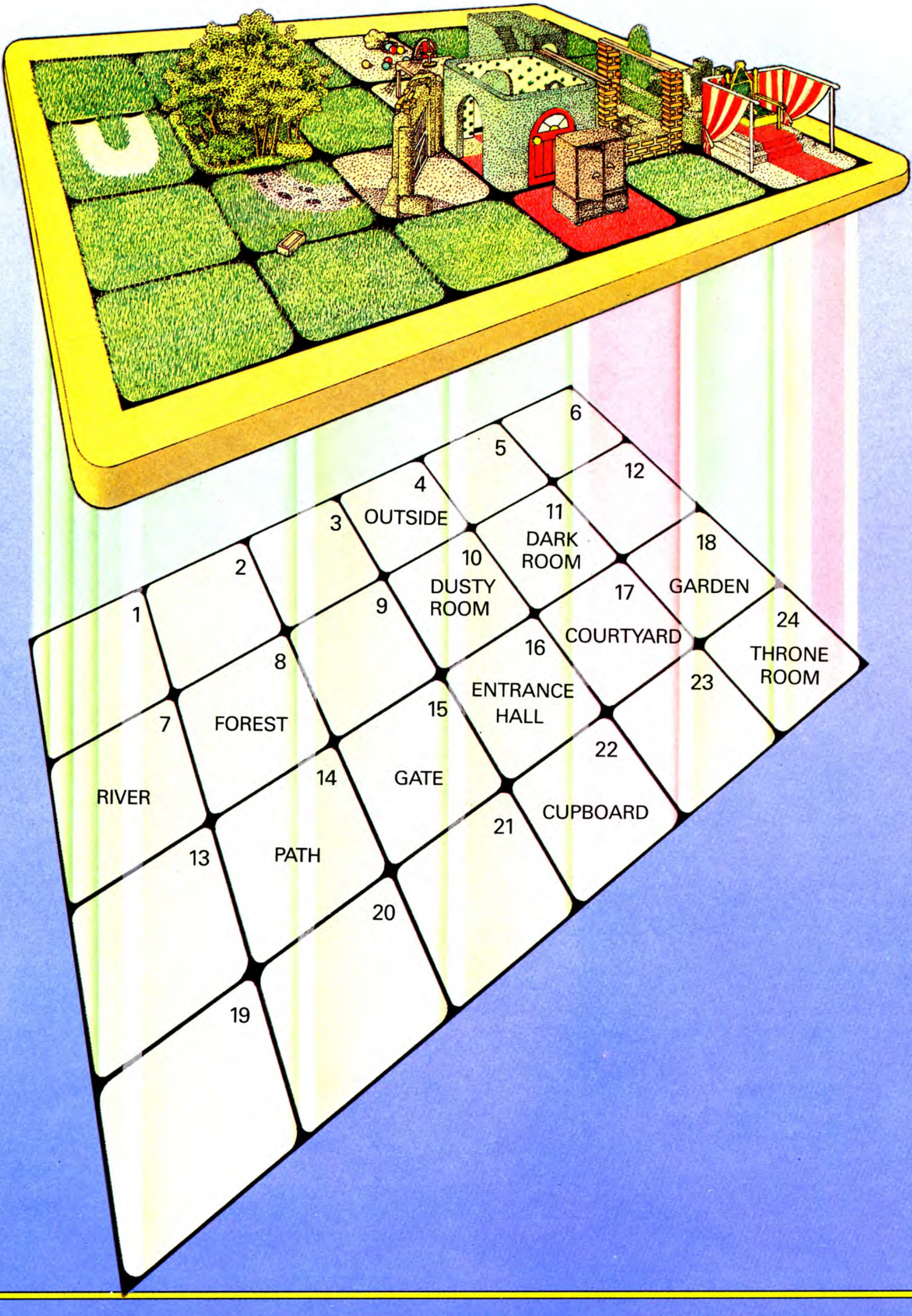
After each location description line, there is another line containing information about the possible exits from the location. The variables N, S, E, and W refer to North, South, East and West. These variables can take one of two values—Ø means that there is no exit in that direction, whilst 1 means that there is an exit.

Finally, there is a RETURN after each section of this program because each location description will be called by a GOSUB.

The extra IF ... THEN in the Dark Room section tests if the adventurer has the lamp, but a description of the variables will be covered later when objects are discussed.

Next time you'll see how to move the adventurer round the locations.

4. The final stage before starting to program. The grid is a direct copy of the map into a form that's easy to work from


1
2
3
4
OUTSIDE
5
6
7
RIVER
8
FOREST
9
10
DUSTY ROOM
11
DARK ROOM
12
13
14
PATH
15
GATE
16
ENTRANCE HALL
17
COURTYARD
18
GARDEN
19
20
21
22
CUPBOARD
23
24
THRONE ROOM

MORE PICTURES FROM MATHS

You have already seen how to use maths functions for circular graphics. Now, look at some more sophisticated uses for SIN and COS, including a working clock

In Part 1 of this series of articles, on pages 250 to 257, you looked at some of the mathematical and angular functions available on your computer. These have already proved themselves to be very valuable tools for a range of graphics applications. Now, let's look closer at how they work, and at some more possible uses for them.

The functions of particular interest at this point are the sine and cosine, and if you are not already familiar with these, you should look through the first article.

In that article, you saw how the functions can be related to the position of a point around the circumference of a circle, and how to use them to work out the coordinates of any point. The compass program on pages 252 and 253 used this to PRINT the numbers marking the directions in their correct positions around the compass face.

DRAWING A CLOCK

So far, none of the programs has done anything that has much practical application, whereas in fact there is a whole range of uses for the trigonometrical functions. Future articles will cover these in more depth, but for the moment, let's have a look at how to draw a clock. You should by now have a very good idea of how one could be drawn.

The principle is very similar to the compass face, except that instead of *you* INPUTing a number which is then shown on the circle, the computer needs to select the angles for itself, increasing with time.

The programs which follow do just that. They calculate the positions of the clock hands, using SIN and COS, to show the correct time.

Although it would be easy enough to draw a normal 12 hour clock, this program draws a 24 hour one which has numbers from 1 to 24 around the face. This is because it gives more opportunity to examine the routines which PRINT the numbers. After you have typed and RUN it we shall look at exactly what SIN and COS are doing.

In the compass program, numbers were printed around the circle to mark the various degrees. The clock program prints numbers around the circle to mark the time, in other words numbers from 1 to 24. These are set up

- DRAWING A CLOCK
- POSITIONING NUMBERS ROUND THE CLOCK FACE
- DRAWING THE HANDS
- ADDING AN ALARM
- ADDING ROMAN NUMERALS TO THE CLOCK
- DRAWING SPIRALS
- USING SIN AND COS TO DRAW MORE ELABORATE PATTERNS

by a FOR ... NEXT loop and PRINTed at coordinates determined by SIN and COS, STEPping to 24 points around the circle.

Part of the Dragon and Tandy program may seem familiar to you if you have used the routine for DRAWing characters on the screen which was given on page 191. It is repeated in this program because the computer cannot PRINT on the screen while in PMODE 4—so it is used to place the numbers around the clockface.

If you SAVEd this routine from the earlier article, you can LOAD it into your computer to avoid having to retype it, then add the extra lines in the program below.

The same routine would also work to add numbers or directions to the compass program covered earlier, and you may like to try adapting the program accordingly.

When the face is drawn, the computer draws a line automatically, showing the second hand of a clock.

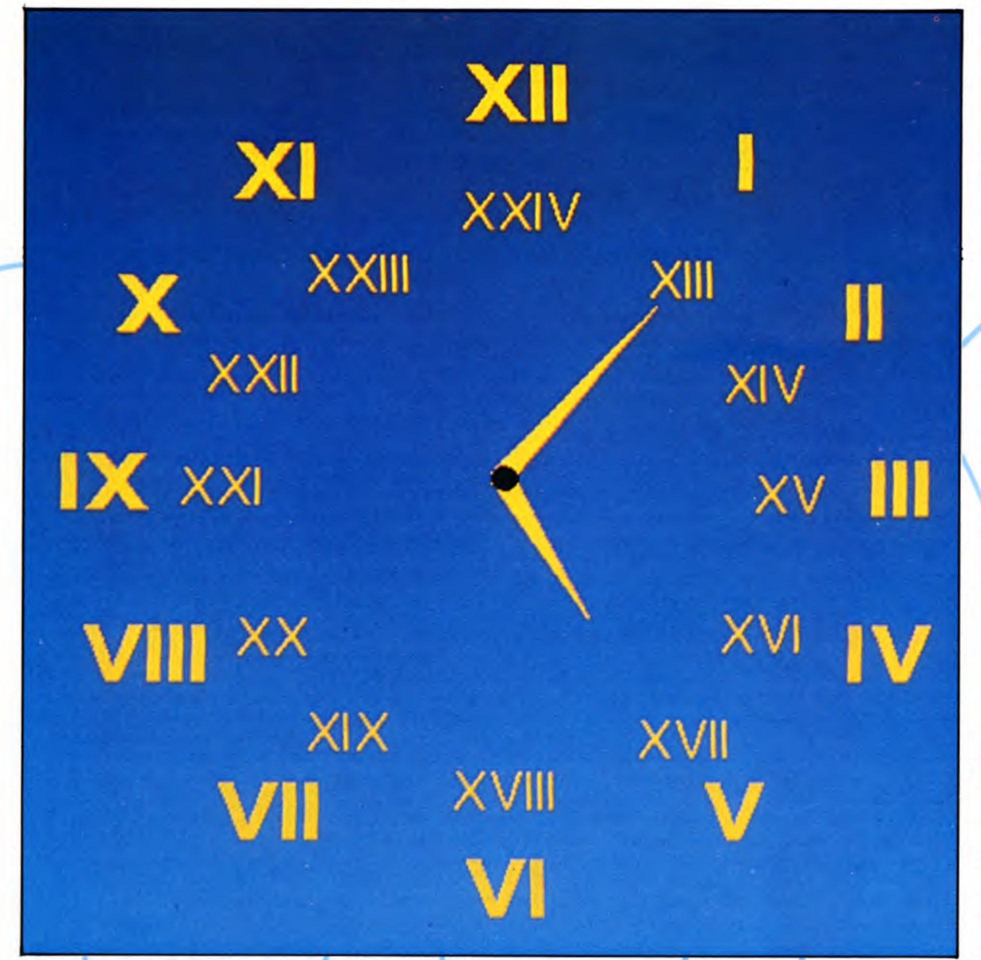
The angle it draws starts at 0 degrees, or the top of the clock face, and gradually gets larger until it has reached 360 degrees, or back to the top again in the same way as the second hand of a clock. The speed at which it STEPs is governed by a PAUSE command or by the timer in the computer.

The clock's hands are moved using a FOR ... NEXT loop on the Commodore and Spectrum. The Acorn, Dragon and Tandy programs use a variable which is updated by the timer each time that the program is run through (the program is a continuous loop). The second hand moves round every time the variable shows that one second has passed.

```
5 CIRCLE 134,92,7Ø
6 LET q= -1
1Ø FOR n=1 TO 24
2Ø PRINT AT 1Ø-1Ø*COS (n/12*PI),
   16+1Ø*SIN (n/12*PI);n
3Ø NEXT n
4Ø FOR t=Ø TO 2ØØØØ
5Ø LET a=t/3Ø*PI
6Ø LET sx=65*SIN a: LET sy=
   65*COS a
7Ø PLOT 134,92: DRAW OVER 1;sx,sy
8Ø PAUSE 42
9Ø PLOT 134,92: DRAW OVER 1;sx,sy
1ØØ NEXT t
```

```
1Ø HIRES Ø,1
2Ø CIRCLE 16Ø,1ØØ,1ØØ,1ØØ,1
3Ø T=Ø
4Ø FOR X=15 TO 36Ø STEP 15:T=T+1
5Ø TEXT 145+9Ø*SIN(X/18Ø*π),98
   -9Ø*COS(X/18Ø*π),STR$(T),1,1,1Ø
6Ø NEXT X
7Ø T=6:T1=357:T2=345:C=12:
   C1=354:C2=33Ø
8Ø A=C/18Ø*π
9Ø LINE 16Ø,1ØØ,6Ø*SIN(A)+16Ø,
   1ØØ-6Ø*COS(A),Ø
1ØØ A=T/18Ø*π
11Ø LINE 16Ø,1ØØ,6Ø*SIN(A)+16Ø,
   1ØØ-6Ø*COS(A),1
12Ø C=T:T=T+6:IF T>36Ø THEN
   T=1:C1=T1:T1=T1+3:IF T1=
   >36Ø THENT1=Ø:C2=T2:T2=T2+15
13Ø IF T2>36Ø THEN T2=.1
14Ø A=C1/18Ø*π
15Ø LINE 16Ø,1ØØ,55*SIN(A)+16Ø,
   1ØØ-55*COS(A),Ø
16Ø A=T1/18Ø*π
17Ø LINE 16Ø,1ØØ,55*SIN(A)+16Ø,
   1ØØ-55*COS(A),1
18Ø A=C2/18Ø*π
19Ø LINE 16Ø,1ØØ,4Ø*SIN(A)+16Ø,
   1ØØ-4Ø*COS(A),Ø
2ØØ A=T2/18Ø*π
21Ø LINE 16Ø,1ØØ,4Ø*SIN(A)+16Ø,
   1ØØ-4Ø*COS(A),1
22Ø GOTO 8Ø
```

```
1Ø GRAPHIC 2
2Ø K=6
3Ø FOR Z=π/12 TO 2*πSTEPπ/12:
   K=K+1:IFK>24THENK=1
4Ø CHAR 1Ø+9*SIN(Z),9+9*COS(Z),
   STR$(K):NEXT
5Ø CHAR 1,9,"24"
6Ø CIRCLE 1,54Ø,5ØØ,351,351
7Ø T=6:T1=357:T2=345:C=12:
   C1=354:C2=33Ø
8Ø A=C/18Ø*π
9Ø DRAW Ø,54Ø,5ØØ TO 2ØØ*SIN(A)
   +54Ø,5ØØ-2ØØ*COS(A)
```



```
100 A=T/180*π
110 DRAW 1,540,500 TO 200*SIN
    (A)+540,500−200*COS(A)
120 C=T:T=T+3:IFT>360THEN
    T=1:C1=T1:T1=T1+3:IFT1
    =>360THENT1=Ø:C2=T2:
    T2=T2+15
130 IF T2>360 THEN T2=.1
140 A=C1/180*π
150 DRAW Ø,540,500 TO 200*SIN
    (A)+540,500−200*COS(A)
160 A=T1/180*π
170 DRAW 1,540,500 TO 200*SIN
    (A)+540,500−200*COS(A)
180 A=C2/180*π
190 DRAW Ø,540,500 TO 150*SIN
    (A)+540,500−150*COS(A)
200 A=T2/180*π
210 DRAW 1,540,500 TO 150*SIN
    (A)+540,500−150*COS(A)
220 GOTO 80
```

```
10 T=1
20 MODE1
30 GCOL3,3
40 MOVE 680,782
50 FOR A=Ø TO 2*PI STEP .Ø5
60 DRAW680+270*COS(A+PI/2),
    512+270*SIN(A+PI/2)
70 NEXT
80 VDU5
90 FOR A=2*PI−PI/12 TO Ø STEP−PI/12
100 MOVE650+310*COS(A+PI/2),
    525+310*SIN(A+PI/2)
110 PRINT;T:T=T+1
120 NEXT
130 T=Ø:TIME=Ø:TM=Ø:TH=Ø
140 MOVE680,512
150 DRAW 680+200*SIN(PI/60*TH),
    512+200*COS(PI/60*TH)
160 MOVE680,512
170 DRAW 680+250*SIN(PI/30*TM),
    512+250*COS(PI/30*TM)
180 MOVE680,512
190 DRAW 680+150*SIN(PI/30*T),
    512+150*COS(PI/30*T)
200 REPEAT
210 MOVE680,512
220 DRAW 680+150*SIN(PI/30*T),
    512+150*COS(PI/30*T)
230 MOVE680,512
250 T=TIME/100
260 DRAW 680+150*SIN(PI/30*T),
    512+150*COS(PI/30*T)
270 IF TIME>6000 THEN PROCMIN:
    TIME=TIME−6000
280 UNTILØ
290 DEF PROCHOUR
300 MOVE680,512
310 DRAW 680+200*SIN(PI/36*TH),
    512+200*COS(PI/36*TH)
320 TH=TH+1
330 MOVE680,512
340 DRAW 680+200*SIN(PI/36*TH),
    512+200*COS(PI/36*TH)
350 ENDPROC
360 DEF PROCMIN
370 MOVE680,512
380 DRAW 680+250*SIN(PI/30*TM),
    512+250*COS(PI/30*TM)
390 TM=TM+1
400 MOVE680,512
410 DRAW 680+250*SIN(PI/30*TM),
    512+250*COS(PI/30*TM)
420 IF (TM MOD 20)=Ø THEN
    PROCHOUR
430 IF TM>59 THEN TM=Ø
440 ENDPROC
```

```
10 PMODE 4,1
20 DIM LE$(26)
30 PCLS
40 FOR K=Ø TO 26:READ LE$(K):NEXT
50 FOR K=Ø TO 9:READ NU$(K):NEXT
60 DATA BR2,ND4R3D2NL3ND2BE2,
    ND4R3DGNL2FDNL3BU4BR2,
    NR3D4R3BU4BR2,ND4R2FD2
    GL2BE4BR,NR3D2NR2D2R3
    BU4BR2
70 DATA NR3D2NR2D2BE4BR,NR3D4R3
    U2LBE2BR,D4BR3U2NL3U2BR2,
    ND4BR2,BD4REU3L2R3BR2,
    D2ND2NF2E2BR2
80 DATA D4R3BU4BR2,ND4FREND4BR2,
    ND4F3DU4BR2,NR3D4R3U4BR2,
    ND4R3D2NL3BE2,NR3D4R3NHU4BR2
90 DATA ND4R3D2L2F2BU4BR2,BD4R3
    U2L3U2R3BR2,RND4RBR2,D4R2
    U4BR2,D3FEU3BR2,D4EFU4BR2
100 DATA DF2DBL2UE2UBR2,DFND2EUBR2,
    R3G3DR3BU4BR2
110 DATA NR2D4R2U4BR2,BDEND4BR2,
    R2D2L2D2R2BU4BR2,NR2BD2NR2B
    D2R2U4BR2,D2R2D2U4BR2,NR2D2
    R2D2L2BE4,D4R2U2L2BE2BR2,
    R2ND4BR2,NR2D4R2U2NL2U2
    BR2,NR2D2R2D2U4BR2
200 MX=127:SX=127:MY=95:
    SY=95
210 SCREEN 1,1
220 CIRCLE(127,95),6Ø,1
230 PI=ATN(1)/15
240 FOR K=5 TO 120 STEP 5
```

```
250 LINE(127+55*SIN(K*PI),95-55
    *COS(K*PI))-(127+59*SIN
    (K*PI),95-59*COS(K*PI)),PSET
260 A$=MID$(STR$(K/5),2):DRAW
    “BM”+STR$(INT(123+68*SIN
    (K*PI)))+“,”+STR$(INT(92
    -68*COS(K*PI)))+“C5S6”:
    GOSUB 1ØØØ
27Ø NEXT K
28Ø IF TIMER<5Ø THEN 28Ø
29Ø TIMER=Ø:T=T+1
3ØØ IF T=864ØØ THEN T=Ø
31Ø H=T/36ØØ
32Ø M=T/6Ø-INT(H)*6Ø
33Ø S=T-INT(M)*6Ø-INT(H)*36ØØ
34Ø LX=SX:LY=SY
35Ø SX=127+45*SIN(S*PI*2):
    SY=95-45*COS(S*PI*2)
36Ø LINE(127,95)-(SX,SY),PSET
37Ø LINE(127,95)-(LX,LY),PRESET
38Ø LINE(127,95)-(MX,MY),PRESET
39Ø MX=127+3Ø*SIN(M*PI*2):MY=
    95-3Ø*COS(M*PI*2)
4ØØ LINE(127,95)-(MX,MY),PSET
41Ø LINE(127,95)-(HX,HY),PRESET
42Ø HX=127+2Ø*SIN(H*PI*5):HY=
    95-2Ø*COS(H*PI*5)
43Ø LINE(127,95)-(HX,HY),PSET
44Ø GOTO 28Ø
1ØØØ FOR M=1 TO LEN(A$)
1Ø1Ø B$=MID$(A$,M,1)
1Ø2Ø IF B$>=“Ø” AND B$<=“9” THEN
   DRAW NU$(VAL(B$)):GOTO 1Ø5Ø
1Ø3Ø IF B$=“ ” THEN N=Ø ELSE
   N=ASC(B$)-64
1Ø4Ø DRAW LE$(N)
1Ø5Ø NEXT
1Ø6Ø RETURN
```

The ZX81 does not have a command to DRAW a line, and so would have to PLOT each point of the clock's hands separately. Since the resolution of its screen is not as high as the other machines', the ZX81 version would be less than good, so there is no version of this program for the ZX81.

Chapter 19 of the ZX81 manual gives a clock program for the ZX81 which you can use as an example, but this clock has no hands. The outline of this clock is just the numbers marking the time: there is no circle around its edge as there is in the programs above.

The Commodore 64 program is in Simons' BASIC, since a version in normal Commodore BASIC would be very complicated; you thus need a Simons' BASIC cartridge plugged in to your machine. This also applies to the other Commodore 64 programs in this article.

The Commodore Vic 20 program is written using the 'Super Expander' cartridge, and will not RUN unless this cartridge is plugged in. This is because the standard graphics commands on the Vic are too complicated to be used for drawing circles easily. This also applies to the other Vic programs in this article.

Each version of this program uses a very similar routine to PRINT the numbers at their correct positions around the clock face, and this routine shows clearly the way that SIN and COS can be used to control PRINTing operations.

This routine takes a form something like:

```
1Ø FOR n=1 TO 24
2Ø PRINT AT(x coordinate of screen
   centre+1Ø*SIN (2*π/n)),y coordinate of
   screen centre-1Ø*COS (2*π/n));n
3Ø NEXT n
```

Each of the computers handles this in a slightly different way, and these may make the routine seem more complicated than it actually is. For example, they may set up an extra variable for the angle, rather than using (2*π/n) in the PRINT AT statement. The routine for the Spectrum follows the same formula as those for the other computers, but looks different. The reason for this is that the Spectrum uses a slightly different method of PRINTing AT a location on the screen: instead of the first number after the command being the x coordinate, it is the y coordinate. What is more, a value of Ø for this number is not, as usual, at the bottom of the screen, but at the top. To see this on the Spectrum, try:

```
PRINT AT Ø,Ø;“a”
```

You should see an 'a' PRINTed at the top left hand corner of the screen.

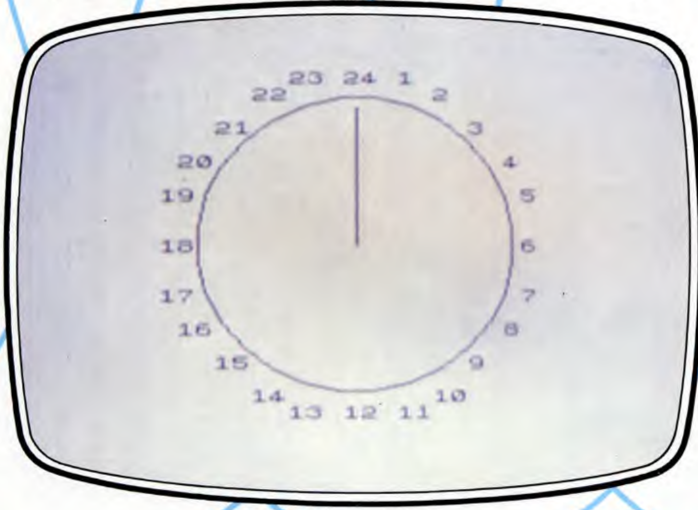
Because of this peculiarity of the Spectrum's PRINT AT command, the COS is the first function in the line: since it is COS which gives the vertical position of the point around the circle's circumference.

If you compare the example above with the actual programs, you will notice that the FOR ... NEXT loop governing the PRINT command is different for each version. In fact this need not be the case, as each different loop would work on any of the computers but the differences were chosen for convenience on each machine's operating system.

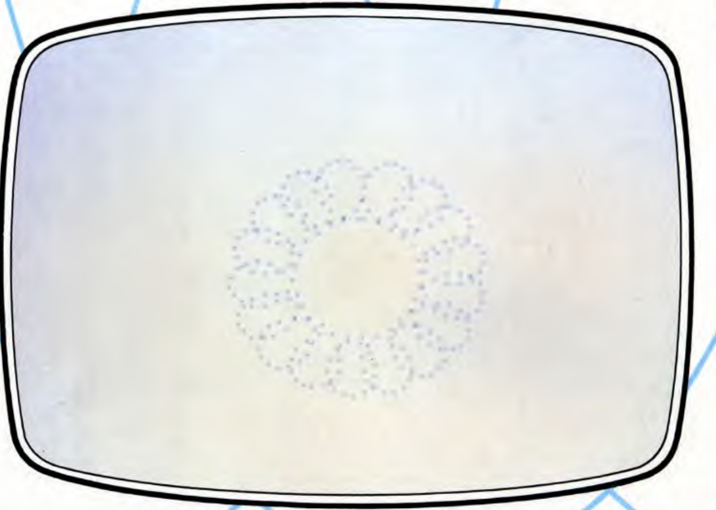
The aim of the routine is to PRINT numbers from 1 to 24 to mark the time around the clock face. So the Spectrum program's loop is

```
FOR n=1 TO 24
NEXT n
```

Within the loop, the routine calculates the PRINT AT positions each time the Spectrum goes through the loop, by dividing a complete circle into 24 segments.



A Spectrum 24 hour clock



Moving spirals on the Spectrum

The Dragon and Tandy do the same as the Spectrum except that their program looks slightly different because they have to use a special routine for DRAWing letters on the graphics screen.

So the Dragon and Tandy do not actually PRINT a number related to the STEP that controls their angular position. Instead, what they PRINT is called up by the command STR$ in the line that controls the PRINTing. STR$ simply refers back to the special DRAWing routine and has nothing to do with where the numbers are drawn, which is controlled by the variable K.

Unlike the loops on these machines, which just go through 24 steps, the Acorn and Commodore loops both involve measurements of a circle.

The Commodore loops (one for the Vic and another for the Commodore 64) are both doing the same thing. The Vic loop is counting in radians, though, while the 64's loop counts in degrees. The Acorn version counts in radians.

As you know, there are 360 degrees in a circle. So to get 24 numbers PRINTed around the clock-face, at equal intervals, the numbers must be placed at 360/24 degree (15°) gaps. In the same way, as there are 2*PI radians in a complete circle, the numbers must be placed at gaps of 2*PI/24, or PI/12 radians.

Note that in all of the loops, the first number is not Ø, but the first STEP up (1 on the Spectrum, 5 on the Dragon, 15 on the Commodore 64, and PI/12 on the Acorn and Vic 20). This is so that the first number will be PRINTed, or on the Dragon, DRAWn, at the 1 o'clock, not the 24 o'clock position.

Since the Commodore and Acorn versions' FOR . . . NEXT loops do not contain the information that is PRINTed, in other words the numbers 1 to 24, but only affect where it is PRINTed, something must be included to tell the computer what to PRINT. So, a variable is used: T on the Acorn and Commodore 64 versions, and K on the Vic 20 program.

Every time that the computer RUNs through this loop, the variable is increased by 1: so that the next number placed on the screen will be the correct next number around the clock-face.

The remainder of the program works in this way using SIN and COS to calculate the positions of the clock hands, and either a variable or the computer's built-in timer to keep track of the time so that the second hand actually moves once every second.

CALLING UP DATA

You can use a different method of telling the computer what to PRINT to the method used in the clock programs: you can store the numbers in a DATA statement and READ them one at a time as the computer goes through the FOR . . . NEXT loop.

While this may seem an unnecessary waste of memory, it leads on to some interesting variations that you can produce.

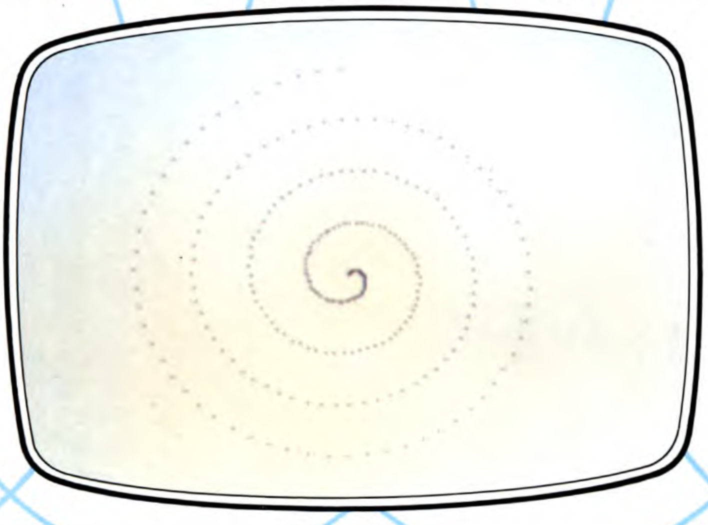
For example, you will probably have seen a clock face with Roman numerals. By using a string variable, and storing letters in a DATA statement instead of numbers, you can change your conventional clock into a more impressive one.

You will probably need to change the size and/or the position of the circle, so that it does not obscure some of the Roman numerals (the number 8 in Roman numerals is VIII, which takes up quite a lot of space!).

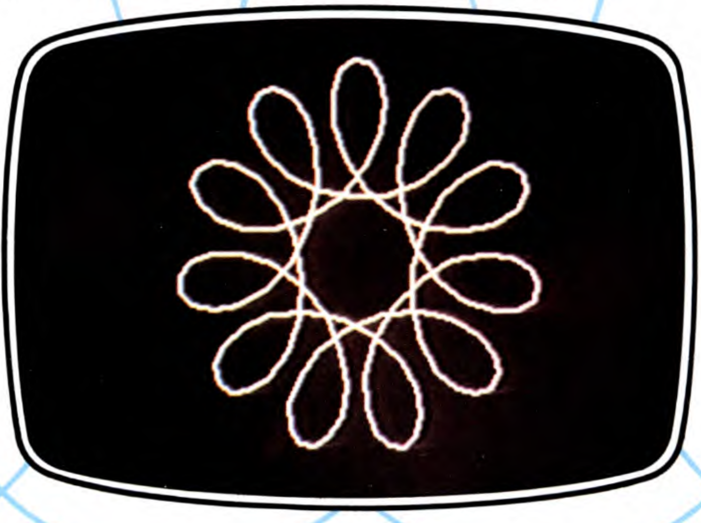
You can also try to make a 24 hour Roman numeral clock. It is surprising how simple it is to change from a 12 hour to a 24 hour clock using this method.

Along similar lines, try to include an alarm into your program. To do this, you will have to work out the position that the clock hands will be in when the set time is reached, using SIN and COS, and add an IF . . . THEN statement so that the computer BEEPs, rings, or chimes as soon as the variables which set the clock hands are equal to the values you have worked out.

Using the same principles, you could make a clock which would cover not just 12 hours or one day, but a week, a month or even a year! The only limit is the amount of information that you can fit onto your screen.



A simply-programmed Spectrum spiral



A flower out of the Acorn screen

DRAWING PATTERNS

The uses you have seen so far for SIN and COS have been quite functional: a clock, a compass, and a graph. But perhaps the most interesting use of them, is the rather more abstract graphics displays that they can be manipulated to achieve.

Just by DRAWing a series of circles and ellipses some very impressive pictures can be DRAWn. But think what would happen if you made a few changes: you could move the centre of the circle, or make it grow each time it goes round, for example. You can achieve changes like this very easily on your computer by just adding extra program lines to the routines you already know.

DRAWING A SPIRAL

Part one of this series of articles gave you a routine for PLOTting pixels to mark the shape of a circle. You can change this a bit and get a spiral instead of a circle. Here are versions of the adapted routine which will draw a spiral pattern for you:

```
1Ø LET z=Ø
5Ø FOR n=Ø TO 8*PI STEP PI/1Ø
6Ø PLOT 128+(5+z)*SIN n,88+
   (5+z)*COS n
7Ø LET z=z+1
8Ø NEXT n
```

```
1Ø HIRES Ø,1
5Ø FOR N=Ø TO 1Ø*π STEP π/1Ø
6Ø PLOT 16Ø+(5+Z)*SIN(N),
   1ØØ+(5+Z)*COS(N),1
7Ø Z=Z+1
8Ø NEXT N
9Ø GOTO 9Ø
```

```
1Ø GRAPHIC 2
5Ø FOR N=Ø TO 1Ø*π STEP π/5Ø
6Ø POINT 1,511+(5+Z)*SIN(N),
   511+(5+Z)*COS(N)
7Ø Z=Z+1
8Ø NEXT N
9Ø GOTO 9Ø
```

```
1Ø MODE1
5Ø FOR T=Ø TO 1Ø*PI STEP.1
6Ø PLOT69,SIN(T)*T*15+68Ø,
   COS(T)*T*15+512
7Ø NEXT
```

```
1Ø PMODE 4,1
2Ø PCLS
3Ø SCREEN 1,1
4Ø PI=4*ATN(1)
5Ø FOR N=Ø TO 1Ø*PI STEP PI/1Ø
6Ø PSET(127+(5+Z)*SIN(N),
   95+(5+Z)*COS(N),5)
7Ø Z=Z+1
8Ø NEXT N
9Ø GOTO 9Ø
```

The only difference between this program and the version that draws a circle is the variable Z. (The Acorn uses a different method.)

This variable starts at Ø, and is increased every time that the computer goes through the loop. Its effect on the program is in the calculations that take place in Line 6Ø.

The PLOTted point is controlled, as in the simple circle, by the SIN and COS of the control variable in the loop. These have to be multiplied by a number so that they are large enough to be noticed on the screen.

So, by making the number by which you multiply the SIN and COS larger each time the computer goes through the loop, you make the position of each pixel get PLOTted further away from the centre than before. The variable which controls this is Z. This results in the spiral growing out gradually from the centre.

As with the circle version of this routine, you can get different results by changing the STEP (in Line 5Ø) or by changing the amount that Z is incremented each time. Some interesting STEPs are: 2,5,PI and 2*PI. Try and work out why the results are different (remember that the computers all work in radians, and that there are 2 times PI radians in a circle).

You can also produce 'spiral ellipses' in the same way that you produced normal ellipses with the first version of this program: by changing the number inside the brackets in Line 6Ø (outside brackets on the Acorns).

MORE ELABORATE PATTERNS

There is a child's toy which involves drawing pretty patterns by using a set of plastic 'wheels'. The child places a small wheel inside a large wheel, and puts a ball point pen through a hole in the small wheel; by moving the small wheel around inside the large one, the pen is moved and draws the patterns on paper.

The centre of the small circle is moving all the time as the child draws so that the line forms a continuous chain of spirals. So the computer can produce similar effects by PLOTting circles whose centre is continuously moving.

The programs below do just this: they PLOT a series of points of a circle, whose centre is continuously moving. And on all the computers, except for the Dragon/Tandy version, the colour is chosen randomly.

Once the program has been RUN it will continue indefinitely. To change the pattern all you need do is press any key (or press ESCAPE if you have an Acorn computer) and the computer will start again.

```
1Ø CLS : LET x=Ø: LET p=Ø:
   LET q=Ø
2Ø LET z=INT (RND*7): INK z:
   LET a=INT (RND*5Ø)
3Ø LET b=INT (RND*5Ø)
4Ø FOR n=Ø TO 2ØØ*PI STEP b/1ØØ
5Ø IF INKEY$< >“” THEN
   GOTO 1Ø
6Ø LET p=128+(a−b)*SIN n:
   LET q=88+(a−b)*COS n
7Ø PLOT p+b*SIN x,q+b*COS x
8Ø LET x=x−a/1Ø74
9Ø NEXT n
11Ø GOTO 2Ø
```

```
1Ø HIRES Ø,1
2Ø A=RND(1)*55:A2=RND(1)*45
3Ø FOR N=Ø TO 2ØØ*π
   STEP A2/9Ø
4Ø P=16Ø+(A−A2)*SIN(N):Q=1ØØ
   +(A−A2)*COS(N)
45 PP=16Ø+(A−A2)*SIN(N−(A2/
   9Ø)):QQ=1ØØ+(A−A2)*COS
   (N−(A2/9Ø))
5Ø LINE PP+A2*SIN(XX), QQ+
   A2*COS(XX),P+A2*SIN(X),
   Q+A2*COS(X),1
6Ø XX=X:X=X−A/19Ø
7Ø IF PEEK(197)< >64 THEN RUN
8Ø NEXT N
9Ø PAUSE 5:RUN
```

```
1Ø GRAPHIC 2:SCNCLR
2Ø A=RND(1)*25Ø:A2=RND(1)*25Ø
3Ø FOR N=Ø TO 2ØØ*π STEP A2/9Ø
4Ø P=511+(A−A2)*SIN(N):Q=511
   +(A−A2)*COS(N)
45 PP=511+(A−A2)*SIN(N−(A2/
   9Ø)):QQ=511+(A−A2)*COS
   (N−(A2/9Ø))
5Ø DRAW2,PP+A2*SIN(XX),QQ+
   A2*COS(XX) TO P+A2*SIN
   (X),Q+A2*COS(X)
6Ø XX=X:X=X−A/19Ø
7Ø IF PEEK(197)< >64 THEN RUN
8Ø NEXT N
9Ø FOR Z=1 TO 2ØØØ:NEXT Z:RUN
```

```
1Ø ON ERROR GOTO 2Ø
2Ø MODE1:VDU19,2,2,Ø,Ø,Ø,19,3,
   4,Ø,Ø,Ø
3Ø X=Ø:GCOLØ,RND(3)
4Ø A=RND(5ØØ):A2=RND(5ØØ)
5Ø MOVE 68Ø,512+A
6Ø FOR N=Ø TO 2ØØ*PI STEP A2/5ØØ
7Ø P=68Ø+(A−A2)*SIN(N):Q=512
   +(A−A2)*COS(N)
8Ø DRAW P+A2*SIN(X),Q+
   A2*COS(X)
9Ø X=X−A/1Ø74
1ØØ NEXT
11Ø GOTO 2Ø
```

```
1Ø PMODE 4,1:PCLS:SCREEN 1,1
2Ø PI=4*ATN(1)
3Ø A=RND(5Ø)−1:B=RND(5Ø)−1
4Ø FOR N=Ø TO 2ØØ*PI STEP
   B/1ØØ
5Ø IF INKEY$< >“” THEN 1Ø
6Ø P=128+(A−B)*SIN(N):Q=
   95+(A−B)*COS(N)
7Ø PSET (P+B*SIN(X),Q+B*
   COS(X),5)
8Ø X=X−A/1Ø74
9Ø NEXT
1ØØ GOTO 3Ø
```

You can see that the FOR ... NEXT loop introduces a circle drawing routine straight away: the loops for each computer are

FOR N=Ø TO 2ØØ * PI

As with the other program which uses SIN and COS in this article, both functions are included inside the loop. They calculate the position of the next pixel to be PLOTted (or, in the case of the Acorn and Commodore computers, DRAWn). Although of course, the calculations are rather more complicated than a simple 'SIN N' or 'COS N'.

The rest of the calculations in this Line are to increase the result of the SIN or COS sum so that it represents a screen position. This is why the centre of each computer's high resolution screen features in the calculations. You should be able to see the 'y' coordinate of the centre of the screen in one half of the program Line (the one which also includes COS), and the 'x' coordinate in the other half.

The other factors in the calculations are two variables. These are given random values at the beginning of each program. It is these random factors which enable the computer to display a different pattern almost every time it RUNs. Try changing the value of these random variables and see what difference it makes.

You can also change the program in another way: change the STEP of the FOR ... NEXT loop. This should make the density of the dots either higher or lower on the Spectrum, Dragon and Tandy, depending on whether you increase or decrease the STEP.

In order to generate the pattern, the centre of the circle is changed after every pixel has been PLOTted. This is done by putting the centre of the circle *itself* on the circumference of another circle which moves round with each step through the loop. This is the function of the second set of SIN and COS calculations, based on x. This variable is reduced very slightly each time that the computer goes through the loop.

Try reducing the variable (x) by a different amount to get another variation on the pattern. These six values give very interesting results: A/1Ø, A/−1Ø, A/−5Ø, A/Ø.5, A/Ø.1, A/Ø.ØØ1.

Each of the computers' versions allow you to press a key while the program is RUNning to start the program again. The Spectrum, Dragon and Tandy use INKEY$ to check for a key-press, while the Commodore PEEKs into its memory to find out the same thing.

On the Acorn computers, you can use the ESCAPE key. This is where the Acorn command 'ON ERROR GOTO ...' becomes very useful. If you press the ESCAPE key the computer records it as an error, and so goes to the Line dictated by the command.

Future articles will cover more uses for the trigonometrical and angular functions, but the next part of the series on maths functions is all about the squares, cubes, square roots and other powers that you can use to control how a variable grows. Once again, these have uses that at first sight would seem to have very little to do with mathematics.

ASSEMBLING BY HAND

- HOW TO WRITE ASSEMBLY LANGUAGE
- WHAT ADDRESSING IS
- HOW THE CODES CONVERT
- WHAT MNEMONICS ARE

Warning: Trying to write machine code programs in hex could damage your brain. But you can do it easily in assembly language and then translate into hexadecimal

Machine code programs are fast, efficient and very effective, but they are extremely difficult to write and debug. Even to the experienced eye they can appear to be a meaningless string of numbers, because instructions, data and addresses all appear as strings of hex digits joined end to end.

The solution is not to write programs in machine code at all. Instead, most machine code programs are written in assembly language, then translated into machine code. Usually this is done by another program called an assembler. But if you don't have an assembler you can do the translation by hand and key in the resulting machine code using your machine code monitor.

ASSEMBLY LANGUAGE

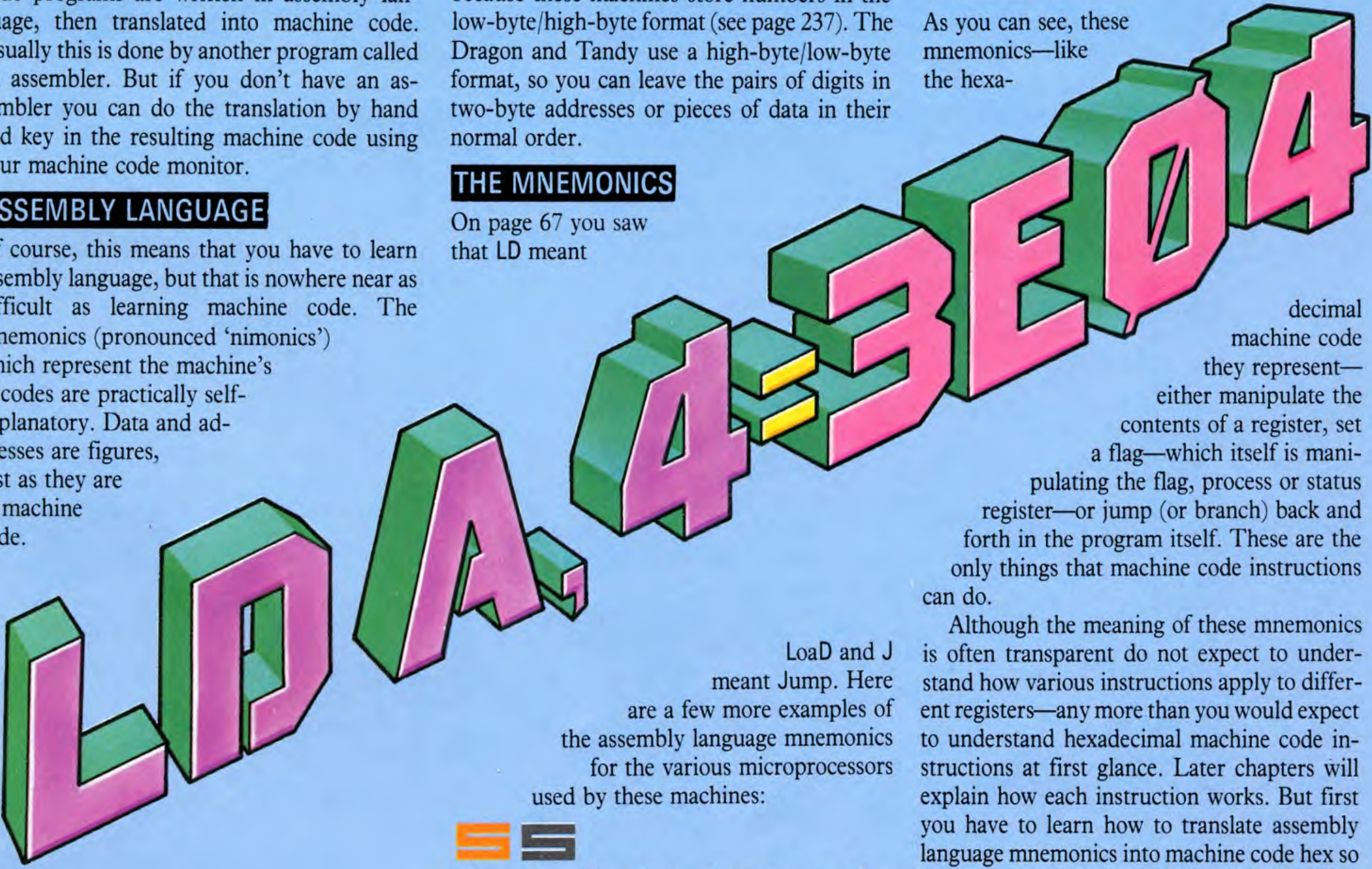
Of course, this means that you have to learn assembly language, but that is nowhere near as difficult as learning machine code. The mnemonics (pronounced 'nimonics') which represent the machine's opcodes are practically self-explanatory. Data and addresses are figures, just as they are in machine code. But with the opcodes in mnemonics, the machine code's solid string of numbers is broken up and you can see what is going on.

When you hand assemble, all you have to do is look up the mnemonics in the appropriate table in the guide to the microprocessor chip in your computer. The Spectrum uses the Z80, the Dragon and Tandy use the 6809 and the Commodore the 6510 (which is almost identical to Acorn's 6502). The result in the table gives you the numerical opcode. Slot that into place and the translation is practically done.

There is only one other thing to remember. On the Sinclairs and the Commodores, you must remember to switch round the bytes of any two-byte address or piece of data. This is because these machines store numbers in the low-byte/high-byte format (see page 237). The Dragon and Tandy use a high-byte/low-byte format, so you can leave the pairs of digits in two-byte addresses or pieces of data in their normal order.

THE MNEMONICS

On page 67 you saw that LD meant LoaD and J meant Jump. Here are a few more examples of the assembly language mnemonics for the various microprocessors used by these machines:



add—add; **call**—call a subroutine; **inc b**—increment (add 1 to) register B; **res**—reset a bit to zero; **jrnz**—jump relative if non-zero; **set**—set a bit to one.

ADC—add with carry; DEX—decrement (subtract 1 from) register X; JSR—jump to subroutine; TAX—transfer data from accumulator to index register X; CMP—compare; BEQ—branch if equal to zero.

ADD—add; JSR—jump to subroutine; INCB—increment accumulator B; BNE—branch if not equal to zero; CLRA—clear contents of accumulator A, that is set A to zero; TFR X,Y—transfer contents of register X to register Y; SEX—extend accumulator B into accumulator A.

As you can see, these mnemonics—like the hexadecimal machine code they represent—either manipulate the contents of a register, set a flag—which itself is manipulating the flag, process or status register—or jump (or branch) back and forth in the program itself. These are the only things that machine code instructions can do.

Although the meaning of these mnemonics is often transparent do not expect to understand how various instructions apply to different registers—any more than you would expect to understand hexadecimal machine code instructions at first glance. Later chapters will explain how each instruction works. But first you have to learn how to translate assembly language mnemonics into machine code hex so that you can convert assembly language listings in *INPUT* and other publications into machine code and feed them into your computer using your machine code monitor.

LD A,(ØE2D

ADDRESSING

Translating assembly language into machine code isn't quite that easy, though. If you look at the instruction set table in a guide to your chip, you will find that a simple instruction like LDA—which means LoaD the A register or accumulator—can be translated into between five and 15 different opcodes, depending on which machine you have. You have to decide on which one of these opcodes to use.

The different opcodes depend on which type of *addressing* is being used. The type of addressing means the various ways information is accessed by the microprocessor.

In this article we shall look at the theory of how this is done on your computer. In part two of this article on hand assembly, you will have a chance to hand assemble a couple of useful assembly language routines yourself, and learn how they work.

The simplest type of addressing with the Z80 chip is *immediate addressing*. Here a number follows the assembly mnemonic. For example:

```
ld a,4
```

This means **load** the A register with the number 4 and translates to 3E Ø4 in machine code. 3E is the code for **ld a** and the data, Ø4, remains the same.

In *direct addressing*, an address where the data can be found is given instead of the data itself. For example:

```
ld a,(ØE2D)
```

means **load** the contents of memory location ØE2D into the A register. This translates into 3A 2D ØE. (Note that the two bytes of the address have been switched round.)

Direct addressing also works the other way round:

```
ld (ØE2D), a
```

means **load** the contents of the A register into memory location ØE2D and it translates into 32 2D ØE. (Again the two bytes are swapped.)

When *indirect addressing* is used, the machine is told where to find the address of the data needed. For example, the instruction:

```
ld a,(hl)
```

means **load** the A register with the data contained in the address in the HL register. In other words, the microprocessor looks in the HL register for the address of the data which it loads into the accumulator.

Again, this works the other way round.

```
ld(hl),a
```

means **load** the contents of the A register into the memory location whose address is in the HL register.

There is a special type of indirect addressing which is called *indexed addressing*. Here one of the two index registers—IX and IY—and the actual address to be used in the operation, is given by an offset (see page 238) which is added to the contents of the IX or IY register.

A typical instruction would be:

```
ld a,(ix+2F)
```

Note that the offset is only one byte.

Data can also be transferred from one register to another. This is called *register-to-register addressing* and its instructions look like this:

```
ld d,b
```

which means **load** the contents of the B register into the D register.

Relative addressing is only used with jump commands which are the equivalent of GOTOs in BASIC. It tells the computer how many bytes to jump forwards or backwards. For example:

```
jrnz ØFC
```

The mnemonic **jrnz** means **j**ump **r**elative if **n**on **z**ero—that is if the zero flag in the flag register is not set (see page 238). The FC after it tells it where to jump to. (The Ø in front of it is to differentiate the hex number FC from any label FC.) FC is −4 in two's complement hex. So if the zero flag is not set, the microprocessor jumps back four bytes in the program.

But the four bytes are counted from the end of the **jrnz** ØFC instruction, which is itself a two-byte instruction—it translates into 2Ø FC in machine code. So the microprocessor jumps back to the instruction which appeared two bytes before this one.

In fact, relative addressing is rarely used in assembly language. Jumps are usually indicated by *labels*. These are words that are used as markers—like '**loop**'—which mark the beginning and end of jumps.

The marker appears in front of the instruction to be jumped to and after the jump instruction. So in an assembly language program you will get a line like:

```
loop ld a,Ø7
```

and later, say:

```
djnz loop
```

The **djnz** means decrement the B register by 1 and jump to where the '**loop**' label appears in front of an instruction if the zero flag is not set. But when you translate into machine code you must work out the relative jumps yourself.

The simplest form of addressing on the 6510 and the 6502 is *implied addressing*. In fact, this is not addressing at all. For example:

```
CLC
```

means CLear Carry flag and does not need any sort of address afterwards. The action is carried out on the carry flag whose address is implied in the instruction.

In *immediate addressing*, the data follows the instruction directly. For example:

```
LDA #&Ø4
```

This LoaDs the Accumulator with the number 4. Look up LDA in the manual for your chip and you will find it translates to A9 when the immediate addressing mode is used. So the whole instruction translates into A9 Ø4.

Absolute addressing is where the whole address of a memory location follows the mnemonic instruction. For example:

```
LDA &1122
```

Q+A

Where do I look up the hex opcodes corresponding to the assembly language mnemonics?

The complete set of machine code instructions is given in Appendix A of your manual under the heading 'The character set'. This listing is somewhat confusing though. And it is difficult to use as the instructions are listed in hex order. You would be better off using one of the programming guides to the Z80 chip or, better still, one dealing with machine code programming specifically on your machine. Check that the book does contain a full list of the opcodes though. Some of them don't.

The Dragon Programmer's Reference Guide and other books on machine code programming on the Dragon and Tandy contain listings of the opcodes. But if you are seriously interested in machine code programming it is probably better to buy one of the programming guides to the 6809 chip.

The Programmer's Reference Guide lists all the opcodes and their assembly language equivalents. But if you are going into machine code programming seriously it is probably worth buying a comprehensive guide to machine code programming on the 6502 or 6510 chips.

means LoaD the A register with the data stored in memory location 1122. This is sometimes known as *direct addressing* on the Acorn machines' 6502 chip.

If you look up LDA under absolute addressing, you will find the opcode AD. And the whole instruction translates into AD 22 11. Note here that the two bytes of the address are swapped around when you translate from assembly language.

On the zero page—that is from ØØØØ to ØØFF—you don't have to specify the first byte of the address, but you have to use a special zero page opcode (called a short address opcode) which tells the computer to look for an address of one byte.

The opcode for LDA in the zero page addressing mode is A5, so an instruction like:

```
LDA &7F
```

will translate to A5 7F.

With both absolute and zero page addresses, it is possible to use *indexed addressing*. Here the contents of one of the index registers—X and Y—are added to the address given with the instruction to give a second address which is to be used. For example:

```
LDA &1122,X
```

Say the contents of X register is 33. The 33 is added to 1122, to give 1155, and the accumulator is loaded with the data found in memory location 1155.

Both the X and Y registers can be used to index both absolute addresses and zero page addresses.

(However you should note that if the sum of the contents of the X register and the zero page address are greater than FF in hex and would therefore fall on page one, the most significant byte is ignored. In zero page addressing, indexed or otherwise, the address used is always on the zero page.)

With indexed addressing you must look up the assembly code mnemonic under 'zero page, X', 'zero page, Y', 'absolute, X' or 'absolute, Y'.

It is also possible to address a piece of data indirectly using *indirect addressing*. This means you give the microprocessor an address—in brackets—where it will find a second address and it is this second address which is used. For example:

```
JMP (&153Ø)
```

means JuMP to the address given at memory location 153Ø. But as any one memory location can hold only one byte and you need two bytes to make up an address, it looks at 153Ø and 1531. The first location holds the least significant byte of the address and the second holds the most significant byte, in accordance with the convention used by these particular microprocessors. So if memory location 153Ø holds 2F and location 1531 holds 13, this means that the microprocessor will jump to memory location 132F.

Indirect addresses can also be indexed in two ways. With the X register you can add an offset to the first address—that is the one given in the instruction. This is called *pre-indexed indirect addressing*. Or you can add an offset from the Y register to the second address—that is the one in the memory locations given in the original instruction. This is called *post-indexed indirect addressing*.

In assembly language the two instructions look like this:

```
LDA (&1122,X) and
LDA (&1122),Y
```

The first is pre-indexed, the second post-indexed and you have to look up the opcode for LDA under (indirect, X) and (indirect), Y. This gives A1 and B1 respectively. So these two instructions translate into A1 22 11 and B1 22 11.

Branch instructions are conditional jumps. For example:

```
BEQ
```

means Branch if EQual—that is, if the zero flag is set to 1. Branch instructions can use *relative addressing* on some assemblers.

BEQ #&Ø4

means jump forward four bytes from the start of the next instruction if the zero flag is set. And

BEQ #&FA

means jump backwards six bytes if the zero flag is set. This is counted backwards from the beginning of the next instruction so two bytes are already taken up by the BEQ instruction. FA is −6 in 2s complement (see page 181).

In assembly language, *labels* are mainly used instead of numbers. These are single word markers which show the microprocessor where to jump to. One instruction will have the label in front of it like this:

label LDA &Ø4

and a branch will have

BEQ label

The assembler will then work out the relative jump. But if you are hand assembling you will have to work out the jump yourself. Machine code does not recognize labels, only numbers.

The 6510 and the 6502 chips have one more type of addressing called *accumulator addressing*. This is used with shift and rotation instructions. For example:

ASL A

means Shift the Accumulator Left one bit. You can also apply this instruction to any memory location—not only the accumulator. In this case you would replace the last A with the address of the memory location you wanted shifted. The memory location can be indexed with the X register only, and it will make the least significant bit of that byte Ø push the most significant bit into the carry flag and shift all the other bits along one place to the left.

Manuals on the 6809 chip used in the Dragon and Tandy draw attention to *inherent addressing* which is also sometimes is known as *implied* or *register addressing*. This simply refers to assembly language instructions which don't need to be followed by an address as they operate on a register which is specified in the instruction itself. For example:

DECA

which means DECrement the A register by 1. Look this up under inherent addressing and you find the code 4A.

In *immediate addressing*, the data follows the instruction directly. For example:

ADDB #$7

which means ADD 7 to the B register. The code for ADDB in immediate mode is CB so the whole thing translates to CB Ø7.

On the Dragon and Tandy there is another form of immediate addressing using a second byte—known as a *postbyte*. In this case, instructions take the form:

TFR A,B

which means transfer the contents of register A into register B.

To translate this instruction into machine code, you look up TFR under immediate addressing (although in some of the manuals it is put, mistakenly, under inherent addressing). This gives 1F. Then the postbyte is evaluated a nibble at a time.

In the manuals for the 6809 chip you will find single-digit values assigned to each of the registers. Two of these are put together to make up the postbyte.

For example, the A register is assigned a value of 8 and the B register a value of 9. So

TFR A,B

translates to 1F 89. If the instruction had been

TFR B,A

the corresponding machine code instruction would be 1F 98.

In *extended* or *absolute addressing*, the full 16-bit address of the memory location where the data required is stored appears after the instruction. For example:

STA $753Ø

which means STore the contents of the Accumulator in memory location &H753Ø. This translates into B7 75 3Ø. (Note that the two bytes of the address are not reversed when writing machine code as they are on most other home computers.)

This is a little cumbersome since the translation—B7 75 3Ø—is a three-byte instruction. Using *direct addressing* this could be cut to a two-byte instruction.

The 6809 chip has a direct page register (see page 239) which stores the most significant byte of an address. So all the memory locations on that page can be addressed using only one byte, the least significant. It works like this.

The direct page register is loaded by using the TFR instruction mentioned above or by the EXG or exchange command which exchanges the contents of any two registers.

EXG A,DP

means EXchanGe the contents of the A register and the Direct Page register. It not only sets the direct page with whatever you have previously loaded into the A register with the LDA command, it also saves the former direct page number by putting the contents of the DP register into the A register. From there it can be stored in a specific memory location by using the STA command.

The most significant byte of the direct page—75, in this example—is now set and the least significant can be given with the instruction. In this case, you have to look up the STA instruction under the direct addressing column, which gives 97. So the full instruction is 97 3Ø.

Obviously, it is not worth going through this whole procedure each time data is stored in a memory location. Setting the direct page

takes much more time than is saved by cutting the three-byte instruction down to a two-byte instruction. But it is often worthwhile setting the direct page at the beginning of the program and saving all the data on the same page.

Indirect addressing takes place when the microprocessor is told to look at one address or register to find a second address where data is to be found or stored. On the Dragon and Tandy the first address or register should be in square brackets. For example,

```
LDA [$FFFE]
```

tells the microprocessor to look in FFFE and the next memory location, FFFF. The two bytes in these memory locations are taken as the address of a second memory location. And it is from this second memory location that the data is taken and loaded into the A register.

Indirect addressing can also be performed via the U, S, X and Y registers and the program counter.

In all these cases of indirect addressing the instruction mnemonic has to be looked up under the *indexed addressing* column to give the opcode. You then have to look up a *postbyte* code.

The postbytes used in indirect addressing appear in a separate table in the manual for the 6809 chip.

In the example given above you look up LDA under indexed addressing and get A6. Then you look up [mmnn]—which indicates a general 16-bit indirect address—in the postbyte table. This gives you 9F (or BF or DF or FF as the postbytes are repeated for each register, but [mmnn] is independent of any register so it appears four times).

So the whole instruction

```
LDA [$FFFE]
```

translates into A6 9F FF FE.

Using the U, S, X and Y registers and the program counter, both *non-indirect* addresses (without square brackets) and indirect addresses (with square brackets) can be given using offsets (see page 239). Offsets can be constants—decimal, eight or 16-bit hex—or the contents of other registers. These are added to, or subtracted from the contents of these five indexable addresses. For example:

```
LDA Ø,X
```

means LoaD the A register with the data in the memory location whose address is given by the contents of the X register. The offset is zero.

```
LDA 1,X
```

means LoaD the A register with the data in the memory location whose address is given by the contents of the X register plus 1.

```
LDA -16,Y
```

means LoaD the A register with the data in the memory location whose address is given by the contents of the Y register minus 16.

```
LDA [$1Ø,X]
```

is the indirect addressing version. It LoaDs the A register with the contents of the memory location whose address is given by the X register plus 1Ø in hex.

In all these cases the opcode for LDA is under the indexed heading and the appropriate postbyte must be added from the postbyte table. 16-bit addresses and eight-bit hex offsets then follow that.

Jumps in assembly language—which are the equivalent of GOTOs and GOSUBs in BASIC—are handled by the commands JMP and JSR. These can have direct page, extended or indexed end addresses and their machine codes alter accordingly. Branches—that is, conditional jumps—use *relative addressing*, though.

With relative addressing you tell the microprocessor how many bytes to jump forward or backwards. There are two main types of branches—an ordinary eight-bit branch which jumps up to 127 bytes forward or 128 bytes backwards and a long 16-bit branch which jumps up to 32,767 bytes forward or 32,768 bytes backwards. For example:

```
BEQ $FA
```

will jump six bytes backwards from the start of the next instruction—all relative jumps are counted from the start of the next instruction—if the zero flag is set. FA is −6 in 2's complement (see page 239). Alternatively, the long branch:

```
LBEQ $ØAØØ
```

jumps 256Ø bytes forward if the zero flag is set.

Branches always use relative addressing—no other type of addressing is available to them. But it is often laborious to work out how big a branch is, and *labels* can be used.

A label is a word which is used to mark the place a branch goes to. For example, you will have a line like:

```
LABEL DECA
```

then later, say:

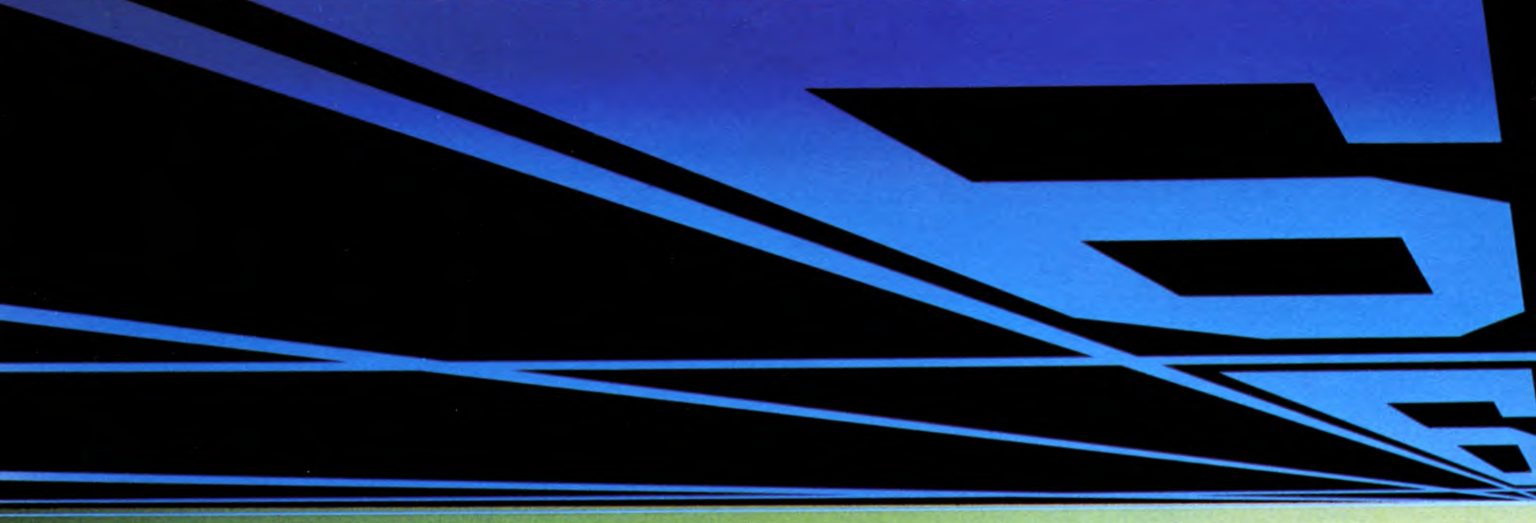
```
BEQ LABEL
```

An assembler would then work out the jump back—or forward if the prefixed LABEL occurs after the branch—to the other mention of LABEL and fill in the number of bytes the program has to jump.

But when you hand assemble you will have to work out these jumps for yourself. You do this by counting the number of bytes from the end of the branch command—that is after the byte that will contain the size of the jump itself—to the beginning of the instruction you are branching to. Remember with a long branch, details of the jump size will take up two bytes.

ASSEMBLING BY HAND

Once you've grasped the principles of addressing, hand assembly is easy. All you have to do is look up the opcode for the assembly mnemonics under the right addressing mode, work out any relative jumps and—on the Spectrum or the Commodore—switch the bytes of any two byte address or piece of data.



UNDERSTANDING ASCII CODES

The ASCII code employed by computers has enough similarities to enable computers literally to 'talk' to each other. This 'common language' has other uses too . . .

Every key and combination of keys on your computer is represented by a unique electronic code pattern within the computer. In BASIC, these patterns are represented by a series of decimal values ranging from 0 to 255. The letter A, for example, has the decimal value 65, B is 66 and so on. And when you type in any letters or words it is these codes that the computer stores.

The values and corresponding pattern of the keys are not the same on every computer, but mercifully there is at least some standardization in the way that they are used and described.

The range of values are referred to by the initials ASCII (which stands for American Standard Code for Information Interchange). This code was devised to provide a means by which computers could transfer data between each other. The Acorns, Commodores, Dragon, Tandy and Spectrum all make use of the code—at least in part. The ZX81 has a completely different code—unique to this machine.

ASCII CODE

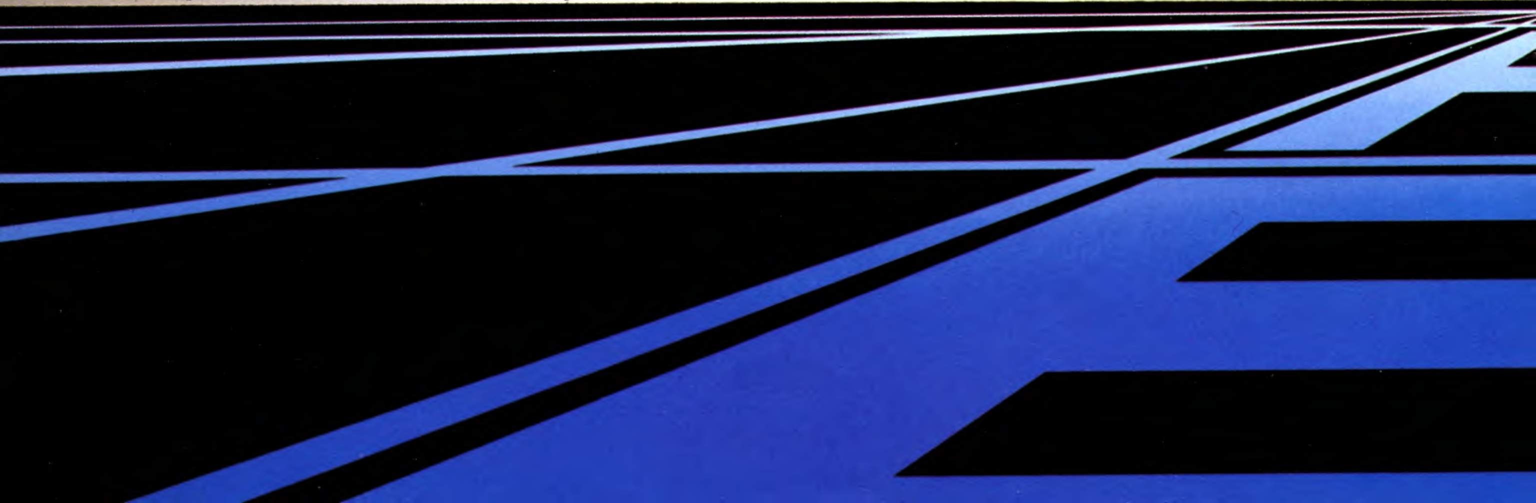
The full ASCII character set is by no means the same on all computers. But there are similarities. The greatest consistency is in the range 33 to 90 which covers normal symbols, puntuation, numerals and the upper case alphabet (capitals). There is an easy way to find out the ASCII code of a character, just type PRINT ASC("X")—or PRINT CODE "X" on the Spectrum—and you'll be shown the ASCII code of "X" or any other character you're interested in. The next program lets you enter the characters more easily.

```
10 PRINT "TYPE ANY LETTER, NUMBER OR
   CHARACTER"
20 INPUT A$
30 PRINT "THE ASCII CODE FOR□";
   A$;"□IS□";ASC(A$)
40 GOTO 20
```

On the Spectrum change Line 30 to:

```
30 PRINT "THE ASCII CODE FOR□";
   A$;"□IS□";CODE A$
```

The reverse of ASC or CODE is CHR$, as this converts the code number into its corresponding character. The following program converts all the ASCII codes between 33 and 90 into characters:



- WHAT IS THE ASCII CODE?
- USING NUMBERS INSTEAD OF CHARACTERS
- A PROGRAM TO PRINT OUT CODED MESSAGES
- THE VALUE OF THE ASCII CODE IN STRING COMPARISONS
- CHECKING INPUT VALUES
- SPECIAL CODE VALUES FOR YOUR COMPUTER

```
10 PRINT "ASCII CODE","CHARACTER"
2Ø FOR N=33 TO 9Ø
3Ø PRINT N,CHR$ (N)
4Ø FOR D=1 TO 5ØØ : NEXT D
5Ø NEXT N
```

The lower case alphabet on all but the Commodore and Vic covers the range from 97 to 122, and this can be checked by changing the last figure in Line 2Ø to 122. The whole set of characters can be displayed by changing it to 255 instead. The full listing of the whole character set is in your manual.

On the Commodore 64 you have to toggle the screen display backwards and forwards between upper case/graphics and upper case/lower case modes by simultaneously pressing the C= and SHIFT keys again and again. This is because the lower case letters and the graphics characters share the same ASCII codes and which you actually see depends on which mode you are in.

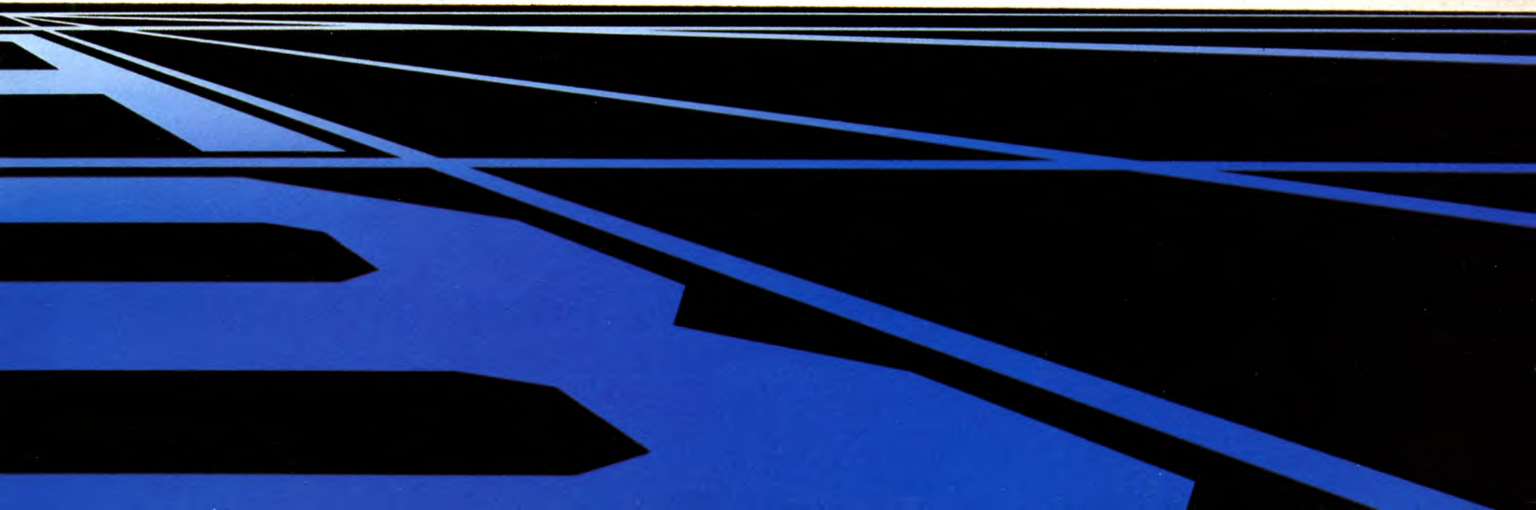
USING THE ASCII CODE

The advantage of using a number instead of a character is that you can alter the number in various mathematical ways. Then, if you print out the CHR$ of the new number you'll get a different letter. This is the basis for many code-writing programs. It's a useful facility, because you cannot directly manipulate a letter by applying a mathematical operation.

Simple codes just add on a constant amount to each number, so in effect the letter is shifted down the alphabet. For example, A becomes G, B becomes H and so on. But these codes are very easy to crack—after all, it wouldn't take long to program a computer to try out each of the 26 combinations, and it would be easy to see at a glance which was the right one.

To be successful, the program has to do more complicated things with the numbers, and the program below makes use of a code word so each letter of the message is altered in different ways. This type of code is very difficult to crack—unless you know the code word that is.

```
1Ø POKE 23658,8: LET CP=Ø: LET PM=Ø:
   LET D$=""
2Ø CLS
3Ø INPUT "WHAT IS THE CODEWORD ?□";
   LINE C$
4Ø PRINT "INPUT THE MESSAGE:—"
```



```
50 INPUT LINE M$
60 LET CP=CP+1: IF CP>LEN C$ THEN
   LET CP=1
70 LET PM=PM+1
80 IF PM>LEN M$ THEN GOTO 200
90 LET F$=M$(PM)
100 IF F$<"A" OR F$>"Z" THEN GOTO
   150
110 LET F=CODE F$+CODE C$(CP)-65
120 IF F>90 THEN LET F=F-26
130 LET D$=D$+CHR$ F
140 GOTO 60
150 IF F$<"Ø" OR F$>"9" THEN LET
   D$=D$+F$:GOTO 70
160 LET F=CODE F$+CODE C$(CP)-48
170 IF F>57 THEN LET F=F-10: GOTO
   170
180 LET D$=D$+CHR$ F
190 GOTO 60
200 PRINT '"THE CODED MESSAGE IS :—"
210 PRINT 'D$
220 STOP
```

```
10 PRINT "♥"
30 INPUT "WHAT IS THE CODEWORD";C$
40 PRINT "INPUT THE MESSAGE :—"
50 INPUT MS$
60 CP=CP+1:IF CP>LEN(C$) THEN CP=1
70 PM=PM+1
80 IF PM>LEN(MS$) THEN 200
90 F$=MID$(MS$,PM,1)
100 IF F$<"A" OR F$>"Z" THEN 150
110 F=ASC(F$)+ASC(MID$
   (C$,CP,1))-65
120 IF F>90 THEN F=F-26
130 CD$=CD$+CHR$(F)
140 GOTO 60
150 IF F$<"Ø" OR F$>"9" THEN
   CD$=CD$+F$:GOTO 70
160 F=ASC(F$)+ASC(MID$
   (C$,CP,1))-48
170 IF F>57 THEN F=F-10: GOTO 170
180 CD$=CD$+CHR$(F)
190 GOTO 60
200 PRINT:PRINT "THE CODED MESSAGE IS
   :—"
210 PRINT:PRINT CD$
220 END
```

```
10 MODE1
30 INPUTLINE"WHAT IS THE
   CODEWORD□",C$
40 PRINT"INPUT THE MESSAGE"
50 INPUTLINE MS$
60 CP=CP+1:IF CP>LEN(C$) THEN CP=1
70 PM=PM+1
80 IF PM>LEN(MS$) THEN 200
90 F$=MID$(MS$,PM,1)
100 IF F$<"A" OR F$>"Z" THEN 150
110 F=ASC(F$)+ASC(MID$(C$,
   CP,1))-65
120 IF F>90 THEN F=F-26
130 CD$=CD$+CHR$(F)
140 GOTO 60
150 IF F$<"Ø" OR F$>"9" THEN
   CD$=CD$+F$:GOTO 70
160 F=ASC(F$)+ASC(MID$(C$,
   CP,1))-48
170 IF F>57 THEN F=F-10:
   GOTO 170
180 CD$=CD$+CHR$(F)
190 GOTO 60
200 PRINT'"THE CODED MESSAGE IS"
210 PRINT'CD$
220 END
```

```
10 CLEAR 300
20 CLS
30 LINEINPUT"WHAT IS THE CODEWORD
   ?";C$
40 PRINT"INPUT THE MESSAGE :—"
50 LINEINPUT MS$
60 CP=CP+1:IF CP>LEN(C$) THEN CP=1
70 PM=PM+1
80 IF PM>LEN(MS$) THEN 200
90 F$=MID$(MS$,PM,1)
100 IF F$<"A" OR F$>"Z" THEN 150
110 F=ASC(F$)+ASC(MID$(C$,
   CP,1))-65
120 IF F>90 THEN F=F-26
130 CD$=CD$+CHR$(F)
140 GOTO 60
150 IF F$<"Ø" OR F$>"9" THEN
   CD$=CD$+F$:GOTO 70
160 F=ASC(F$)+ASC(MID$(C$,
   CP,1))-48
170 IF F>57 THEN F=F-10:  GOTO 170
180 CD$=CD$+CHR$(F)
190 GOTO 60
200 PRINT:PRINT"THE CODED
   MESSAGE IS :"
210 PRINT:PRINTCD$
220 END
```

The program asks you to INPUT the codeword and the message, then it prints out the coded version. The message is now safe and can only be decoded by someone who also knows the codeword.

The way it works is quite simple and the method is much the same as a simple coding program, but instead of adding a constant amount to each letter, the ASCII code of the first letter of the codeword is added to the first letter of the message, the second letter of the codeword is added to the second letter of the message, and so on. When the end of the codeword is reached, the cycle starts at its beginning again.

If when the numbers are added on the result is greater than 90—that is beyond Z—then 26 is subtracted to keep the range within the alphabet. Numbers are treated separately, in Lines 15Ø to 18Ø, to keep their ASCII codes between 48 and 57 which corresponds to the numbers Ø to 9.

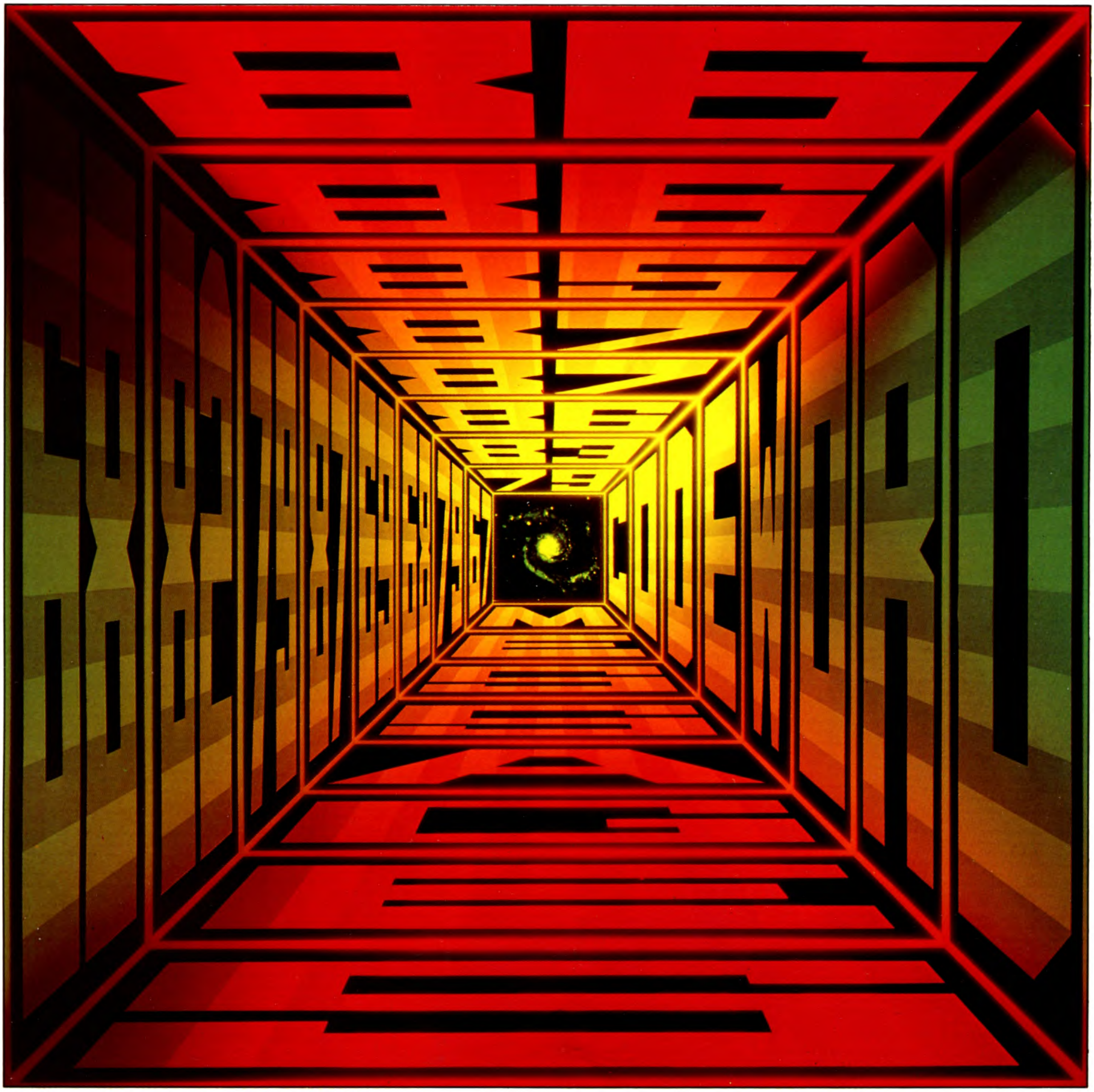
The decoding program is very similar to the first one, all that's different is that a few +s and −s have been changed. The Line numbers follow on from the last one so you can save the combined program, and there are a few extra Lines to let you choose whether you want to code or decode a message:

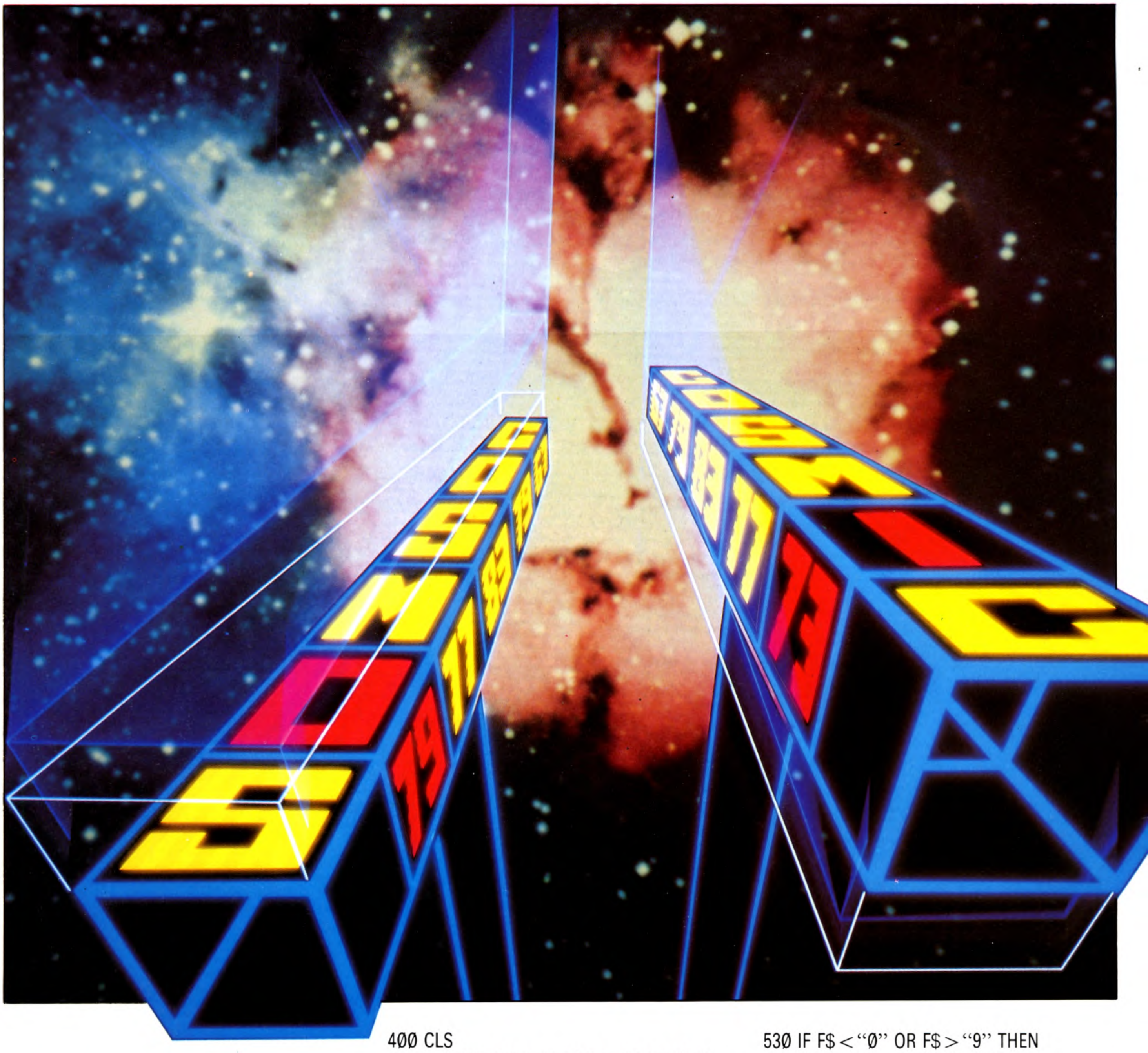
```
12 INPUT "(C)ODE OR (D)ECODE ?";
   LINE A$
14 IF A$="D" THEN GOTO 400
16 IF A$<>"C" THEN GOTO 12
400 CLS
410 INPUT "WHAT IS THE CODEWORD ?□";
   LINE C$
420 PRINT "INPUT THE CODED MESSAGE:—"
430 INPUT LINE M$
440 LET CP=CP+1: IF CP>LEN C$ THEN
   LET CP=1
450 LET PM=PM+1
460 IF PM>LEN M$ THEN GOTO 580
470 LET F$=M$(PM)
480 IF F$<"A" OR F$>"Z" THEN GOTO
   530
490 LET F=CODE F$-CODE C$(CP)+65
500 IF F<65 THEN LET F=F+26
510 LET D$=D$+CHR$ F
520 GOTO 440
530 IF F$<"Ø" OR F$>"9" THEN LET
   D$=D$+F$: GOTO 450
540 LET F=CODE F$-CODE C$(CP)+48
550 IF F<48 THEN LET F=F+10:GOTO550
560 LET D$=D$+CHR$ F
570 GOTO 440
580 PRINT '"THE DECODED MESSAGE IS:—"
590 PRINT 'D$
600 STOP
```

```
12 INPUT "(C)ODE OR (D)ECODE";A$
14 IF A$="D" THEN GOTO 400
16 IF A$<>"C" THEN GOTO 12
400 PRINT"♥"
410 INPUT "WHAT IS THE CODEWORD";C$
420 PRINT "INPUT THE CODED MESSAGE :"
430 INPUT MS$
440 CP=CP+1:IF CP>LEN(C$) THEN CP=1
450 PM=PM+1
460 IF PM>LEN(MS$) THEN 580
470 F$=MID$(MS$,PM,1)
480 IF F$<"A" OR F$>"Z" THEN 530
490 F=ASC(F$)-ASC(MID$(C$,
   CP,1))+65
```

```
500 IF F<65 THEN F=F+26
510 CD$=CD$+CHR$(F)
520 GOTO 44Ø
530 IF F$<"Ø" OR F$>"9" THEN
   CD$=CD$+F$:GOTO 45Ø
540 F=ASC(F$)-ASC(MID$(C$,
   CP,1))+48
550 IF F<48 THEN F=F+1Ø:GOTO 55Ø
560 CD$=CD$+CHR$(F)
570 GOTO 44Ø
580 PRINT "THE DECODED
   MESSAGE IS :—"
590 PRINT:PRINT CD$
600 END
```

```
12 INPUT "(C)ODE OR (D)ECODE",A$
14 IF A$="D" THEN GOTO 4ØØ
16 IF A$<>"C" THEN GOTO 12
4ØØ CLS
41Ø INPUTLINE"WHAT IS THE
   CODEWORD□",C$
42Ø PRINT"INPUT THE CODED MESSAGE"
43Ø INPUTLINE MS$
44Ø CP=CP+1:IF CP>LEN(C$) THEN
   CP=1
45Ø PM=PM+1
46Ø IF PM>LEN(MS$) THEN 58Ø
47Ø F$=MID$(MS$,PM,1)
48Ø IF F$<"A" OR F$>"Z" THEN 53Ø
49Ø F=ASC(F$)-ASC(MID$(C$,
   CP,1))+65
5ØØ IF F<65 THEN F=F+26
51Ø CD$=CD$+CHR$(F)
52Ø GOTO 44Ø
53Ø IF F$<"Ø" OR F$>"9" THEN
```





```
   CD$ = CD$ + F$:GOTO 45Ø
54Ø F = ASC(F$) – ASC(MID$(C$,
   CP,1)) + 48
55Ø IF F < 48 THEN F = F + 1Ø:GOTO 55Ø
56Ø CD$ = CD$ + CHR$(F)
57Ø GOTO 44Ø
58Ø PRINT'"THE DECODED MESSAGE IS"
59Ø PRINT'CD$
6ØØ END
```

```
12 INPUT "(C)ODE OR (D)ECODE ?";A$
14 IF A$ = "D" THEN GOTO 4ØØ
16 IF A$ < > "C" THEN GOTO 12
```

```
4ØØ CLS
41Ø LINEINPUT"WHAT IS THE CODEWORD
   ?";C$
42Ø PRINT"INPUT THE CODED MESSAGE :—"
43Ø LINEINPUT MS$
44Ø CP = CP + 1:IF CP > LEN(C$) THEN
   CP = 1
45Ø PM = PM + 1
46Ø IF PM > LEN(MS$) THEN 58Ø
47Ø F$ = MID$(MS$,PM,1)
48Ø IF F$ < "A" OR F$ > "Z" THEN 53Ø
49Ø F = ASC(F$) – ASC(MID$(C$,
   CP,1)) + 65
5ØØ IF F < 65 THEN F = F + 26
51Ø CD$ = CD$ + CHR$(F)
52Ø GOTO 44Ø
53Ø IF F$ < "Ø" OR F$ > "9" THEN
   CD$ = CD$ + F$:GOTO 45Ø
54Ø F = ASC(F$) – ASC(MID$(C$,
   CP,1)) + 48
55Ø IF F < 48 THEN F = F + 1Ø:GOTO 55Ø
56Ø CD$ = CD$ + CHR$(F)
57Ø GOTO 44Ø
58Ø PRINT"THE DECODED MESSAGE IS :—"
59Ø PRINT:PRINTCD$
6ØØ END
```

By the way, if you really want to make the code difficult to crack you can always code your coded message using a second codeword! Remember to get them in the right order when you decode the message again.

COMPARISONS

One of the major uses of the ASCII code values is for comparing strings (see page 202). String comparisons are made one character at a time from left to right until the end of one word is reached. For example, suppose two string constants such as "COSMOS" and "COSMIC" have to be evaluated. The computer first compares the two Cs, then the Os, Ss and Ms. Then O is compared to the corresponding character I on the other side.

What is compared is not an arbitrary set of alphabetic values, but the ASCII value of each character. O has a greater ASCII value than I so in this comparison "COSMOS" has a greater value than "COSMIC". Note that the sums of the ASCII values of the letters which make up the string are *not* used to determine which has a greater value. Each letter is compared individually. Have a look at the article on pages 201 to 207 for more examples of how strings are compared.

Here are some example comparisons showing various possible forms of A$ and B$, along with the ASCII values of the characters involved. It is important to compare the values of each character on a pair by pair basis, the first character of A$ with the first of B$ and so on for the length of the shorter word.

A$/ASCII		B$/ASCII		Relationship
ABC	65,66,67	ABC	65,66,67	A$ = B$
ABD	65,66,68	ABCD	65,66,67	A$ > B$
ABD	65,66,68	ABCD	65,66,67	A$ < > B$
ABC	65,66,67	Abc	65,97,98	A$ < B$
COSMI	67,79,83, 77,73	COSMO	67,79,83, 77,79	A$ < B$
$1	36,49	$1.Ø	36,49,46	A$ < B$

Note, in the second and third examples, it is possible to write the relationship in more than one way. Also that, in the last example, that all else being equal, it's the longer string which is considered to have the greater value or be 'more than' the other.

CHECKING INPUTS

The ASC and CODE functions work only on the first character of the string. So PRINT ASC("USA") or PRINT CODE"USA" (on the Spectrum) gives 85, the ASCII code of "U". This turns out to be very useful for checking INPUTs to programs. For example, in the code-writing program, Line 12 asked you to INPUT either C for CODE or D for DECODE. Then the next two lines direct the program to the right section. But to guard against someone typing in CODE or DECODE in full you could rewrite Lines 14 and 16 as:

```
14 IF ASC(A$) = 68 THEN GOTO 4ØØ
16 IF ASC(A$) < > 67 THEN GOTO 12
```

```
14 IF CODE A$ = 68 THEN GOTO 4ØØ
16 IF CODE A$ < > 67 THEN GOTO 12
```

CONTROL CODES

Some ASCII codes don't have characters associated with them. For example, when you press the ENTER or RETURN key the computer stores the value 13, but instead of printing a character on the screen, the cursor is moved to the beginning of the next line or, on the Spectrum, the program is listed. Code 13 is called the carriage return code or, sometimes, the newline code. Try typing PRINT "A";CHR$(13);"B" and you'll see that "B" is printed on a new line directly below the "A".

Here's a way to find out the ASCII codes of the non-character keys:

```
1Ø PRINT CODE INKEY$
2Ø GOTO 1Ø
```

```
5 GET A$ : IF A$ = ""THEN 5
1Ø PRINT ASC(A$)
2Ø GOTO 5
```

```
5 A$ = INKEY$ : IF A$ = "" THEN 5
1Ø PRINT ASC(A$)
2Ø GOTO 1Ø
```

```
1Ø PRINT ASC GET$
2Ø GOTO 1Ø
```

Try pressing ENTER or RETURN, DELETE, backspace or any of the other keys on your computer. You can use this method to detect when the keys are pressed in a game. Unfortunately, except on the Acorn computers, you cannot use the values in reverse. That is, although cursor down has code 1Ø, PRINT CHR$(1Ø) will not move the cursor down.

Some computers make more use of these codes than others. There are a lot of spare numbers available—from 1 to 31 and some above 90 but they were originally defined in the days when computers were connected to printers so the codes are mostly concerned with controlling these old-style printers. Now, of course, computers are designed to be used with a TV screen so most of the old codes no longer apply although a few of the codes can be used with modern printers.

The Spectrum computer has redefined a lot of the codes so they work on the TV screen. For example PAPER has code 17 and INK has code 16. So to PRINT the word "TITLE" on the screen in red on a green background you can use this program:

```
1Ø LET A$ = CHR$17 + CHR$4 + CHR$16 +
   CHR$2 + "TITLE"
2Ø PRINT A$
```

If this was a heading you wanted to use several times in a program then you just need to define A$ once, at the start, and then use PRINT A$ each time.

The Commodores make extensive use of the code values above and below the punctuation and alphabet range. The values used up to code 32 (space) include several of the available colours, some cursor controls, and a forced switch to lower case, amongst others. One of the most commonly used values from the early range of codes is CHR$(13) which represents RETURN.

The remaining colours and cursor controls, plus the ROM graphics, can be found in the code value range which follows 90 (for Z). Codes between 191 and 255 simply duplicate this latter range.

The codes can be concatenated to give variables which have embedded cursor and colour controls. Look, for example, at the datafile listing on page 48, Lines 42 to 6Ø. You can check the meaning of the various code values using the relevant User's Guide appendix. For example, code 14 in Line 42 forces the display into lower case when the program is RUN. The variable names used in that program suggest the functions performed by the code values assigned to them. Note how a variable such as c4$ can be formed from cd$ which is previously defined using CHR$(17). As you can see, a very wide range of screen manipulations can be carried out.

The Acorn computers have redefined a lot of the codes and altered others so they work on the TV screen. For example, code 22 selects the screen mode, 17 defines the text colour and 31 moves the cursor to a specified position. The advantage of using CHR$22 instead of MODE, and CHR$17 instead of COLOUR is that the character strings can be concatenated together into one long string which you can then call by a single variable name.

So, using the last three examples try out the following program:

```
1Ø A$ = CHR$22 + CHR$5 + CHR$17 +
   CHR$2 + CHR$31 + CHR$8 +
   CHR$15 + “TITLE”
2Ø PRINT A$
```

If this was a heading that you wanted printed several times in a program then you just need to define A$ once and use PRINT A$ each time.

This use of control codes, though, is unusual. PRINT CHR$(X); is equivalent to VDU X. This means less typing and you can also list several control codes together in a single VDU statement. The last program is the same as:

```
1Ø VDU 22,5,17,2,31,8,15
2Ø PRINT “TITLE”
```

You could even put the word “TITLE” into the VDU statement—just list the ASCII codes of each letter:

```
1Ø VDU22,5,17,2,31,8,15,84,73,84,
   76,69
```

However, this may be going a bit too far! These control codes all have equivalent keywords that do the same thing—MODE, COLOUR and so on, but not all control codes are like this. For example if you want to redefine the colours you have to use VDU19, there is no word that does the same. Similarly VDU7 makes a short beep and VDU1Ø (or PRINT CHR$1Ø;) moves the cursor down one space. The whole list of VDU codes is in the User Guide.

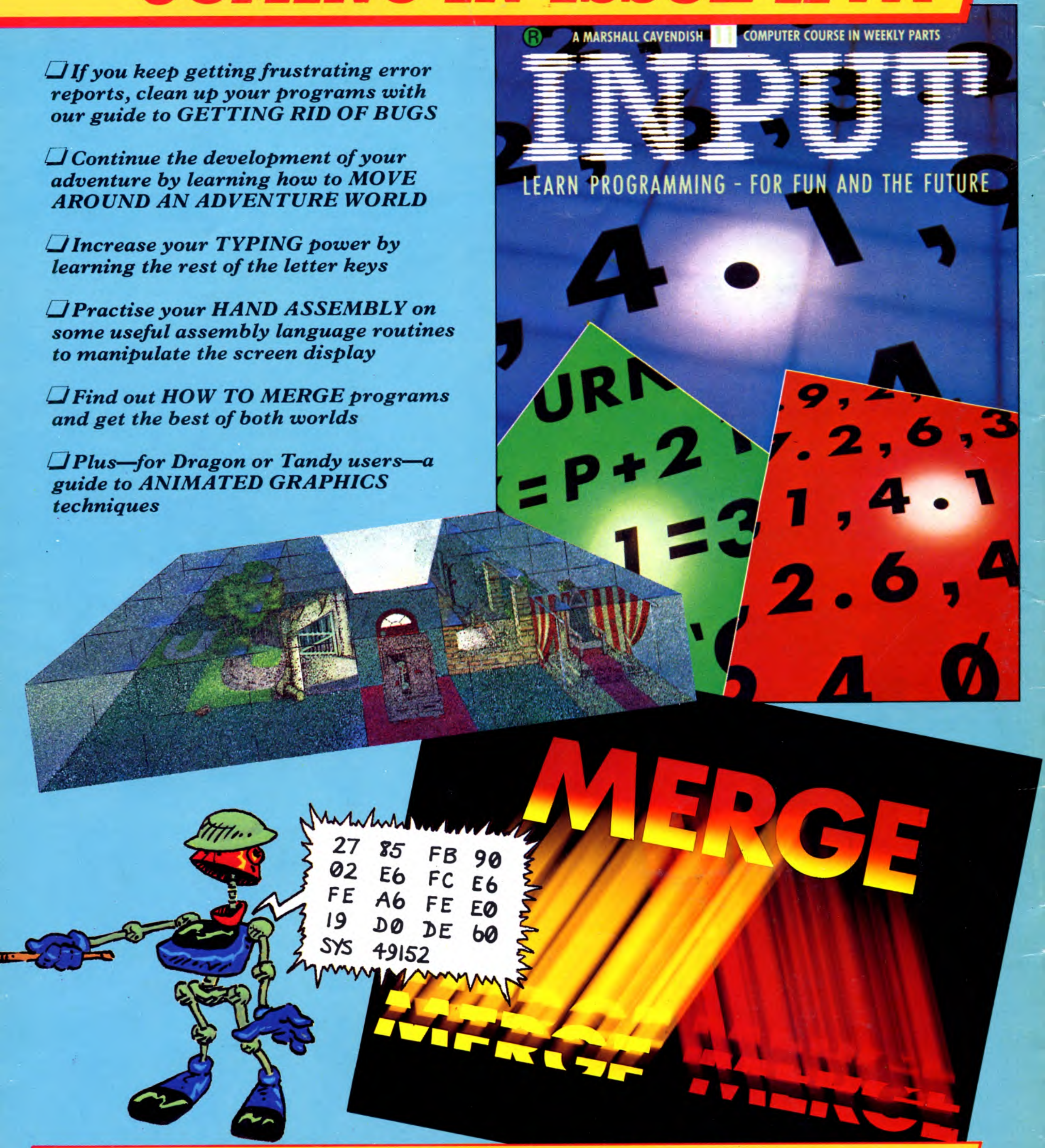
CUMULATIVE INDEX

An interim index will be published each week. There will be a complete index in the last issue of *INPUT*.

The publishers accept no responsibility for unsolicited material sent for publication in INPUT. All tapes and written material should be accompanied by a stamped, self-addressed envelope.

COMING IN ISSUE 11 . . .

- ❑ *If you keep getting frustrating error reports, clean up your programs with our guide to GETTING RID OF BUGS*
- ❑ *Continue the development of your adventure by learning how to MOVE AROUND AN ADVENTURE WORLD*
- ❑ *Increase your TYPING power by learning the rest of the letter keys*
- ❑ *Practise your HAND ASSEMBLY on some useful assembly language routines to manipulate the screen display*
- ❑ *Find out HOW TO MERGE programs and get the best of both worlds*
- ❑ *Plus—for Dragon or Tandy users—a guide to ANIMATED GRAPHICS techniques*


®
A MARSHALL CAVENDISH 11 COMPUTER COURSE IN WEEKLY PARTS
INPUT
LEARN PROGRAMMING - FOR FUN AND THE FUTURE
MERGE
MERGE
MERGE

```
27  85  FB  90
02  E6  FC  E6
FE  A6  FE  E0
19  D0  DE  60
SYS  49152
```

ASK YOUR NEWSAGENT FOR INPUT